

CMF
DEC
DEC
DEC
DEC

[illegible]

```

LL          IIIIII          SSSSSSSS
LL          IIIIII          SSSSSSSS
LL          II             SS
LL          II             SS
LL          II             SS
LL          II             SS
LL          II             SSSSSS
LL          II             SSSSSS
LL          II             SS
LL          II             SS
LL          II             SS
LL          II             SS
LLLLLLLLLLLL IIIIII          SSSSSSSS
LLLLLLLLLLLL IIIIII          SSSSSSSS

```

(2)	72	Declarations
(4)	133	VAX\$SUBP6 - Subtract Packed (6 Operand Format)
(5)	184	VAX\$ADDP6 - Add Packed (6 Operand Format)
(6)	239	VAX\$SUBP4 - Subtract Packed (4 Operand Format)
(7)	282	VAX\$ADDP4 - Add Packed (4 Operand Format)
(8)	336	ADDPx/SUBPx Common Initialization Code
(9)	457	ADD_PACKED - Add Two Packed Decimal Strings
(10)	615	ADD_PACKED_BYTE - Add Two Bytes Containing Decimal Digits
(11)	722	SUBTRACT_PACKED - Subtract Two Packed Decimal Strings
(12)	924	SUB_PACKED_BYTE - Subtract Two Bytes Containing Decimal Digits
(13)	1040	STORE_RESULT - Store Decimal String
(14)	1125	VAX\$MULP - Multiply Packed
(15)	1322	Common Exit Path for VAX\$MULP and VAX\$DIVP
(16)	1500	EXTEND_STRING_MULTIPLY - Multiply a String by a Number
(17)	1582	VAX\$DIVP - Divide Packed
(18)	1938	QUOTIENT_DIGIT - Get Next Digit in Quotient
(19)	2100	MULTIPLY_STRING - Multiply a String by a Number
(20)	2154	DECIMAL_ROPRAND
(21)	2195	ARITH_ACCVIO - Reflect an Access Violation
(22)	2245	Access Violation Handling for ADDPx and SUBPx
(23)	2319	Access Violation Handling for MULP and DIVP

```
0000 1 .TITLE VAX$DECIMAL_ARITHMETIC - VAX-11 Packed Decimal Arithmetic Instructio
0000 2 .IDENT /V04-000/
0000 3
0000 4
0000 5 *****
0000 6
0000 7 * COPYRIGHT (c) 1978, 1980, 1982, 1984 BY
0000 8 * DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.
0000 9 * ALL RIGHTS RESERVED.
0000 10
0000 11 * THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
0000 12 * ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
0000 13 * INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
0000 14 * COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
0000 15 * OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
0000 16 * TRANSFERRED.
0000 17
0000 18 * THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
0000 19 * AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
0000 20 * CORPORATION.
0000 21
0000 22 * DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
0000 23 * SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
0000 24
0000 25 *****
0000 26
0000 27
0000 28
0000 29 ++
0000 30 Facility:
0000 31
0000 32 VAX-11 Instruction Emulator
0000 33
0000 34 Abstract:
0000 35
0000 36 The routines in this module emulate the VAX-11 packed decimal
0000 37 instructions that perform arithmetic operations. These procedures can
0000 38 be a part of an emulator package or can be called directly after the
0000 39 input parameters have been loaded into the architectural registers.
0000 40
0000 41 The input parameters to these routines are the registers that
0000 42 contain the intermediate instruction state.
0000 43
0000 44 Environment:
0000 45
0000 46 These routines run at any access mode, at any IPL, and are AST
0000 47 reentrant.
0000 48
0000 49 Author:
0000 50
0000 51 Lawrence J. Kenah
0000 52
0000 53 Creation Date
0000 54
0000 55 19 October 1983
0000 56
0000 57 Modified by:
```

0000 58 :
0000 59 :
0000 60 :
0000 61 :
0000 62 :
0000 63 :
0000 64 :
0000 65 :
0000 66 :
0000 67 :
0000 68 :
0000 69 :
0000 70 :--

V01-003 LJK0037 Lawrence J. Kenah 17-Jul-1984
Fix two minor bugs in exception handling code that caused
MULP and DIVP tests to generate spurious access violations.

V01-002 LJK0024 Lawrence J. Kenah 21-Feb-1984
Add code to handle access violations. Perform minor cleanup.
Eliminate double use of R10 in MULP and DIVP.

V01-001 LJK0008 Lawrence J. Kenah 19-Oct-1983
The emulation code for ADDP4, ADDP6, SUBP4, SUBP6, MULP and
DIVP was moved into a separate module.

```
0000 72      .SUBTITLE      Declarations
0000 73
0000 74      ; Include files:
0000 75
0000 76      .NOCROSS        ; No cross reference for these
0000 77      .ENABLE        SUPPRESSION ; No symbol table entries either
0000 78
0000 79      ADDP4_DEF        ; Bit fields in ADDP4 registers
0000 80      ADDP6_DEF        ; Bit fields in ADDP6 registers
0000 81      DIVP_DEF         ; Bit fields in DIVP registers
0000 82      MULP_DEF         ; Bit fields in MULP registers
0000 83      SUBP4_DEF        ; Bit fields in SUBP4 registers
0000 84      SUBP6_DEF        ; Bit fields in SUBP6 registers
0000 85
0000 86      $PSLDEF          ; Define bit fields in PSL
0000 87      $SRMDEF          ; Define arithmetic trap codes
0000 88
0000 89      .DISABLE        SUPPRESSION ; Turn on symbol table again
0000 90      .CROSS            ; Cross reference is OK now
0000 91
0000 92      ; Symbol definitions
0000 93
0000 94      :               The architecture requires that R4 be zero on completion of an ADDP6 or
0000 95      :               SUBP6 instruction. If we did not have to worry about restarting
0000 96      :               instructions after an access violation, we could simply zero the saved
0000 97      :               R4 value on the code path that these two instructions have in common
0000 98      :               before they merge with the ADDP4 and SUBP4 routines. The ability to
0000 99      :               restart requires that we keep the original R4 around at least until no
0000 100      :              more access violations are possible. To accomplish this, we store the
0000 101      :              fact that R4 must be cleared on exit in R11, which also contains the
0000 102      :              evolving condition codes. We use bit 31, the compatibility mode bit
0000 103      :              because it is nearly impossible to enter the emulator with CM set.
0000 104
0000001F 0000 105      ADD_SUB_V_ZERO_R4 = PSL$V_CM
0000 106
0000 107      ; External declarations
0000 108
0000 109      .DISABLE              GLOBAL
0000 110
0000 111      .EXTERNAL -
0000 112      DECIMAL$BOUNDS_CHECK,-
0000 113      DECIMAL$BINARY_TO_PACKED_TABLE,-
0000 114      DECIMAL$PACKED_TO_BINARY_TABLE,-
0000 115      DECIMAL$STRIP_ZEROS_R0_RT,-
0000 116      DECIMAL$STRIP_ZEROS_R2_R3
0000 117
0000 118      .EXTERNAL -
0000 119      VAX$DECIMAL_EXIT,-
0000 120      VAX$DECIMAL_ACCVIO,-
0000 121      VAX$REFLECT_TRAP,-
0000 122      VAX$ROPRAND
0000 123
0000 124      ; PSECT Declarations:
0000 125
0000 126      .DEFAULT              DISPLACEMENT , WORD
0000 127
00000000 128      .PSECT _VAX$CODE PIC, USR, CON, REL, LCL, SHR, EXE, RD, NOWRT, LONG
```

VAX\$DECIMAL_ARITHMETIC
V04-000

- VAX-11 Packed Decimal Arithmetic Instr 16-SEP-1984 01:33:44 VAX/VMS Macro V04-00 Page 4
Declarations 5-SEP-1984 00:44:34 [EMULAT.SRC]VAXARITH.MAR;1 (2)

0000 129
0000 130

BEGIN_MARK_POINT

```
0000 132
0000 133 .SUBTITLE VAX$SUBP6 - Subtract Packed (6 Operand Format)
0000 134
0000 135 :+ Functional Description:
0000 136
0000 137 In 6 operand format, the subtrahend string specified by the subtrahend
0000 138 length and subtrahend address operands is subtracted from the minuend
0000 139 string specified by the minuend length and minuend address operands.
0000 140 The difference string specified by the difference length and difference
0000 141 address operands is replaced by the result.
0000 142
0000 143 Input Parameters:
0000 144
0000 145 R0 - sublen.rw Number of digits in subtrahend string
0000 146 R1 - subaddr.ab Address of subtrahend string
0000 147 R2 - minlen.rw Number of digits in minuend string
0000 148 R3 - minaddr.ab Address of minuend string
0000 149 R4 - diflen.rw Number of digits in difference string
0000 150 R5 - difaddr.ab Address of difference string
0000 151
0000 152 Output Parameters:
0000 153
0000 154 R0 = 0
0000 155 R1 = Address of the byte containing the most significant digit of
0000 156 the subtrahend string
0000 157 R2 = 0
0000 158 R3 = Address of the byte containing the most significant digit of
0000 159 the minuend string
0000 160 R4 = 0
0000 161 R5 = Address of the byte containing the most significant digit of
0000 162 the string containing the difference
0000 163
0000 164 Condition Codes:
0000 165
0000 166 N <- difference string LSS 0
0000 167 Z <- difference string EQL 0
0000 168 V <- decimal overflow
0000 169 C <- 0
0000 170
0000 171 Register Usage:
0000 172
0000 173 This routine uses all of the general registers. The condition codes
0000 174 are recorded in R11 as the routine executes.
0000 175 :-
0000 176
0000 177 .ENABLE LOCAL_BLOCK
0000 178
0000 179 VAX$SUBP6::
0000 180 PUSH R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11 ; Save the lot
0000 181 MOVZBL #1,R9 ; Indicate that this is subtraction
0000 182 BRB 10$ ; Merge with ADDP6 code
```

OFFF 8F BB 0000 180
59 01 9A 0004 181
06 11 0007 182

```
0009 184 .SUBTITLE VAX$ADDP6 - Add Packed (6 Operand Format)
0009 185
0009 186 Functional Description:
0009 187
0009 188 In 6 operand format, the addend 1 string specified by the addend 1
0009 189 length and addend 1 address operands is added to the addend 2 string
0009 190 specified by the addend 2 length and addend 2 address operands. The sum
0009 191 string specified by the sum length and sum address operands is replaced
0009 192 by the result.
0009 193
0009 194 Input Parameters:
0009 195
0009 196 R0 - add1len.rw Number of digits in first addend string
0009 197 R1 - add1addr.ab Address of first addend string
0009 198 R2 - add2len.rw Number of digits in second addend string
0009 199 R3 - add2addr.ab Address of second addend string
0009 200 R4 - sumlen.rw Number of digits in sum string
0009 201 R5 - sumaddr.ab Address of sum string
0009 202
0009 203 Output Parameters:
0009 204
0009 205 R0 = 0
0009 206 R1 = Address of the byte containing the most significant digit of
0009 207 the first addend string
0009 208 R2 = 0
0009 209 R3 = Address of the byte containing the most significant digit of
0009 210 the second addend string
0009 211 R4 = 0
0009 212 R5 = Address of the byte containing the most significant digit of
0009 213 the string containing the sum
0009 214
0009 215 Condition Codes:
0009 216
0009 217 N <- sum string LSS 0
0009 218 Z <- sum string EQL 0
0009 219 V <- decimal overflow
0009 220 C <- 0
0009 221
0009 222 Register Usage:
0009 223
0009 224 This routine uses all of the general registers. The condition codes
0009 225 are recorded in R11 as the routine executes.
0009 226
0009 227
0009 228 VAX$ADDP6::
0009 229 PUSH R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11 ; Save the lot
0009 230 CLRL R9 ; This is addition
0009 231 10$: ROPRND CHECK R4 ; Insure that R4 is LEQU 31
0009 232 MOVPSL R11 ; Get initial PSL
0009 233
0009 234 ; Indicate that the saved R4 must be cleared on the exit path
0009 235
0009 236 BBCS #ADD_SUB_V_ZERO_R4,R11,30$ ; Set bit and join common code
0009 237 BRB 30$ ; In case we drop through BBCS
```

```
OFFF 8F BB 0009 229
      59 D4 000D 230
      SB DC 000F 231
      001A 232
      001C 233
      001C 234
      001C 235
1D 5B 1F E3 001C 236
      1B 11 0020 237
```

```
0022 239 .SUBTITLE VAX$$SUBP4 - Subtract Packed (4 Operand Format)
0022 240 :+
0022 241 : Functional Description:
0022 242 :
0022 243 : In 4 operand format, the subtrahend string specified by subtrahend
0022 244 : length and subtrahend address operands is subtracted from the difference
0022 245 : string specified by the difference length and difference address
0022 246 : operands and the difference string is replaced by the result.
0022 247 :
0022 248 : Input Parameters:
0022 249 :
0022 250 : R0 - sublen.rw          Number of digits in subtrahend string
0022 251 : R1 - subaddr.ab       Address of subtrahend decimal string
0022 252 : R2 - diflen.rw       Number of digits in difference string
0022 253 : R3 - difaddr.ab      Address of difference decimal string
0022 254 :
0022 255 : Output Parameters:
0022 256 :
0022 257 : R0 = 0
0022 258 : R1 = Address of the byte containing the most significant digit of
0022 259 :       the subtrahend string
0022 260 : R2 = 0
0022 261 : R3 = Address of the byte containing the most significant digit of
0022 262 :       the string containing the difference
0022 263 :
0022 264 : Condition Codes:
0022 265 :
0022 266 : N <- difference string LSS 0
0022 267 : Z <- difference string EQL 0
0022 268 : V <- decimal overflow
0022 269 : C <- 0
0022 270 :
0022 271 : Register Usage:
0022 272 :
0022 273 : This routine uses all of the general registers. The condition codes
0022 274 : are recorded in R11 as the routine executes.
0022 275 : -
0022 276 :
0022 277 VAX$$SUBP4::
OFFF 8F BB 0022 278 PUSHR #^M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11> ; Save the lot
59 01 9A 0026 279 MOVZBL #1,R9 ; Indicate that this is subtraction
06 11 0029 280 BRB 20$ ; Merge with ADDP4 code
```

```
002B 282 .SUBTITLE VAX$ADDP4 - Add Packed (4 Operand Format)
002B 283
002B 284 *+
002B 285 Functional Description:
002B 286 In 4 operand format, the addend string specified by the addend length
002B 287 and addend address operands is added to the sum string specified by the
002B 288 sum length and sum address operands and the sum string is replaced by
002B 289 the result.
002B 290
002B 291 Input Parameters:
002B 292
002B 293 R0 - addlen.rw Number of digits in addend string
002B 294 R1 - addaddr.ab Address of addend decimal string
002B 295 R2 - sumlen.rw Number of digits in sum string
002B 296 R3 - sumaddr.ab Address of sum decimal string
002B 297
002B 298 Output Parameters:
002B 299
002B 300 R0 = 0
002B 301 R1 = Address of the byte containing the most significant digit of
002B 302 the addend string
002B 303 R2 = 0
002B 304 R3 = Address of the byte containing the most significant digit of
002B 305 the string containing the sum
002B 306
002B 307 Condition Codes:
002B 308
002B 309 N <- sum string LSS 0
002B 310 Z <- sum string EQL 0
002B 311 V <- decimal overflow
002B 312 C <- 0
002B 313
002B 314 Register Usage:
002B 315
002B 316 This routine uses all of the general registers. The condition codes
002B 317 are recorded in R11 as the routine executes.
002B 318 :-
002B 319
002B 320 VAX$ADDP4::
002B 321 OFFF 8F BB 002B 321 PUSHB #^M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11> ; Save the lot
002B 322 59 D4 002F 322 CLRL R9 ; This is addition
002B 323
002B 324 ; The output string, described by R4 and R5, will be the same as the input
002B 325 ; string for ADDP4 and SUBP4. It is necessary to explicitly clear R4<31:16>
002B 326 ; along this code path so MOVQ R2,R4 will not always work.
002B 327
002B 328 54 52 3C 0031 328 20$: MOVZWL R2,R4 ; Set output size equal to input size
002B 329 55 53 D0 0034 329 MOVL R3,R5 ; ... and ditto for string addresses
002B 330 5B DC 0037 330 MOVPSL R11 ; Get initial PSL
002B 331
002B 332 ; Indicate that the saved R4 will be restored on the common exit path
002B 333
002B 334 00 5B 1F E5 0039 334 BBCC #ADD_SUB_V_ZERO_R4,R11,30$ ; Clear bit and join common code
```

```
003D 336 .SUBTITLE ADDPx/SUBPx Common Initialization Code
003D 337
003D 338 All four routines converge at this point and execute common initialization
003D 339 code until a later decision is made to do addition or subtraction.
003D 340
003D 341 R4 - Number of digits in destination string
003D 342 R5 - Address of destination string
003D 343
003D 344 R9 - Indicates whether operation is addition or subtraction
003D 345 0 => addition
003D 346 1 => subtraction
003D 347
003D 348 R11<31> - Indicates whether this is a 4-operand or 6-operand instruction
003D 349 0 => 4-operand (restore saved R4 on exit)
003D 350 1 => 6-operand (set R4 to zero on exit)
003D 351
003D 352
5B 04 00 04 F0 003D 353 30$: INSV #PSLSM_Z,#0,#4,R11 ; Set Z-bit, clear the rest in saved PSW
0042 354 ESTABLISH HANDLER - ; Store address of access
0042 355 ARITH_ACCVIO ; violation handler
0047 356
0047 357 ROPRAND CHECK R2 ; Insure that R2 is LEQU 31
004F 358 MARK_POINT ADD SUB BSBW 0
004F 359 BSBW- DECIMAL$STRIP_ZEROS_R2_R3 ; Strip high order zeros from R2/R3
0052 360
0052 361 ROPRAND CHECK R0 ; Insure that R0 is LEQU 31
005A 362 MARK_POINT ADD SUB BSBW 0
005A 363 BSBW- DECIMAL$STRIP_ZEROS_R0_R1 ; Strip high order zeros from R0/R1
005D 364
005D 365 : Rather than totally confuse the already complicated logic dealing with
005D 366 : different length strings in the add or subtract loop, we will put the
005D 367 : result into an intermediate buffer on the stack. This buffer will be long
005D 368 : enough to handle the worst case so that the addition loop need only concern
005D 369 : itself with the lengths of the two input loops. The required length is 17
005D 370 : bytes, to handle an addition with a carry out of the most significant byte.
005D 371 : We will allocate 20 bytes to maintain whatever alignment the stack has.
005D 372
005D 373 CLRQ -(SP) ; Set aside space for output string
005F 374 CLRQ -(SP) ; Worst case string needs 16 bytes
0061 375 CLRL -(SP) ; Add slack for a CARRY
58 54 04 01 EF 0063 376 EXTZV #1,#4,R4,R8 ; Get byte count of destination string
7E 55 55 58 C1 0068 377 ADDL3 R8,R5,-(SP) ; Save high address end of destination
55 18 AE 9E 006C 378 MOVAB 24(SP),R5 ; Point R5 one byte beyond buffer
0070 379
0070 380 : The number of minus signs will determine whether the real operation that we
0070 381 : perform is addition or subtraction. That is, two plus signs or two minus
0070 382 : signs will both result in addition, while a plus sign and a minus sign will
0070 383 : result in subtraction. The addition and subtraction routines have their own
0070 384 : methods for determining the correct sign of the result.
0070 385
0070 386 : For the purpose of counting minus signs, we treat subtraction as the
0070 387 : addition of the negative of the input operand. That is, subtraction of a
0070 388 : positive quantity causes the sign to be remembered as minus and counted as
0070 389 : a minus sign while subtraction of a minus quantity stores a plus sign and
0070 390 : counts nothing.
0070 391
0070 392 : On input to this code sequence, R9 distinguished addition from subtraction.
```

```

0070 393 : On output, it contains either 0, 1, or 2, indicating the total number of
0070 394 : minus signs, real or implied, that we counted.
0070 395
56 50 04 01 EF 0070 396 EXTZV #1,#4,R0,R6 : Get byte count for first input string
51 56 C0 0075 397 ADDL R6,R1 : Point R1 to byte containing sign
0078 398 MARK POINT ADD SUB 24
56 61 F0 8F 8B 0078 399 BICB3 #B11110000,(R1),R6 : R6 contains the sign 'digit'
10 59 E8 007D 400 BLBS R9,35$ : Use second CASE if subtraction
0080 401
0080 402 : This case statement is used for addition
0080 403
0080 404 CASE R6,TYPE=B,LIMIT=#10,<- : Dispatch on sign,digit
0080 405 50$,- : 10 => sign is '+'
0080 406 40$,- : 11 => sign is '-'
0080 407 50$,- : 12 => sign is '+'
0080 408 40$,- : 13 => sign is '-'
0080 409 50$,- : 14 => sign is '+'
0080 410 50$,- : 15 => sign is '+'
0080 411 >
0090 412
0090 413 : This case statement is used for subtraction
0090 414
0090 415 35$: CASE R6,TYPE=B,LIMIT=#10,<- : Dispatch on sign digit
0090 416 40$,- : 10 => treat sign as '-'
0090 417 50$,- : 11 => treat sign as '+'
0090 418 40$,- : 12 => treat sign as '-'
0090 419 50$,- : 13 => treat sign as '+'
0090 420 40$,- : 14 => treat sign as '-'
0090 421 40$,- : 15 => treat sign as '-'
0090 422 >
00A0 423
59 01 D0 00A0 424 40$: MOVL #1,R9 : Count a minus sign
56 0D 9A 00A3 425 MOVZBL #13,R6 : The preferred minus sign is 13
05 11 00A6 426 BRB 60$ : Now check second input sign
00A8 427
59 59 D4 00A8 428 50$: CLRL R9 : No real minus signs so far
56 0C 9A 00AA 429 MOVZBL #12,R6 : The preferred minus sign is 12
00AD 430
57 52 04 01 EF 00AD 431 60$: EXTZV #1,#4,R2,R7 : Get byte count for second input string
53 57 C0 00B2 432 ADDL R7,R3 : Point R3 to byte containing sign
00B5 433 MARK POINT ADD SUB 24
57 63 F0 8F 8B 00B5 434 BICB3 #B11110000,(R3),R7 : R7 contains the sign 'digit'
00BA 435
00BA 436 CASE R7,TYPE=B,LIMIT=#10,<- : Dispatch on sign,digit
00BA 437 80$,- : 10 => sign is '+'
00BA 438 70$,- : 11 => sign is '-'
00BA 439 80$,- : 12 => sign is '+'
00BA 440 70$,- : 13 => sign is '-'
00BA 441 80$,- : 14 => sign is '+'
00BA 442 80$,- : 15 => sign is '+'
00BA 443 >
00CA 444
59 59 D6 00CA 445 70$: INCL R9 : Remember that sign was minus
57 0D 9A 00CC 446 MOVZBL #13,R7 : The preferred minus sign is 13
03 11 00CF 447 BRB 90$ : Now check second input sign
00D1 448
57 0C 9A 00D1 449 80$: MOVZBL #12,R7 : The preferred minus sign is 12
```

03 59	E9	00D4	450	90S:	BLBC	R9,ADD_PACKED	; Even parity indicates addition
		00D4	451				
		00D7	452				
00B3	31	00D7	453		BRW	SUBTRACT_PACKED	; Odd parity calls for subtraction
		00DA	454				
		00DA	455		.DISABLE	LOCAL_BLOCK	

00DA 457
00DA 458
00DA 459
00DA 460
00DA 461
00DA 462
00DA 463
00DA 464
00DA 465
00DA 466
00DA 467
00DA 468
00DA 469
00DA 470
00DA 471
00DA 472
00DA 473
00DA 474
00DA 475
00DA 476
00DA 477
00DA 478
00DA 479
00DA 480
00DA 481
00DA 482
00DA 483
00DA 484
00DA 485
00DA 486
00DA 487
00DA 488
00DA 489
00DA 490
00DA 491
00DA 492
00DA 493
00DA 494
00DA 495
00DA 496
00DA 497
00DA 498
00DA 499
00DA 500
00DA 501
00DA 502
00DA 503
00DA 504
00DA 505
00DA 506
00DA 507
00DA 508
00DA 509
00DA 510
00DA 511
00DA 512
00DA 513

.SUBTITLE ADD_PACKED - Add Two Packed Decimal Strings

Functional Description:

This routine adds two packed decimal strings whose descriptors are passed as input parameters and places their sum into another (perhaps identical) packed decimal string.

At the present time, the result is placed into a 16-byte storage area while the sum is being evaluated. This drastically reduces the number of different cases that must be dealt with as each pair of bytes in the two input strings is added.

The signs of the two input strings have already been dealt with so this routine performs addition in all cases, even if the original entry was at SUBP4 or SUBP6. The cases that arrive in this routine are as follows.

	R2/R3	R0/R1	result
R2/R3 + R0/R1	plus	plus	plus
R2/R3 + R0/R1	minus	minus	minus
R2/R3 - R0/R1	minus	plus	minus
R2/R3 - R0/R1	plus	minus	plus

Note that the correct choice of sign in all four cases is the sign of the second input string, the one described by R2 and R3.

Input Parameters:

- R0<4:0> - Number of digits in first input decimal string
R1 - Address of least significant digit of first input decimal string (the byte containing the sign)
- R2<4:0> - Number of digits in second input decimal string
R3 - Address of least significant digit of second input decimal string (the byte containing the sign)
- R4<4:0> - Number of digits in output decimal string
R5 - Address of one byte beyond least significant digit of intermediate string stored on the stack
- R6<3:0> - Sign of first input string in preferred form
R7<3:0> - Sign of second input string in preferred form

```
00DA 514 : R11 - Saved PSL (Z-bit is set, other condition codes are clear)
00DA 515 :
00DA 516 : (SP) - Saved R5, address of least significant digit of ultimate
00DA 517 : destination string.
00DA 518 : 4(SP) - Beginning of 20-byte buffer to hold intermediate result
00DA 519 :
00DA 520 : Output Parameters:
00DA 521 :
00DA 522 : The particular input operation (ADDPx or SUBPx) is completed in
00DA 523 : this routine. See the routine headers for the four routines that
00DA 524 : request addition or subtraction for a list of output parameters
00DA 525 : from this routine.
00DA 526 :
00DA 527 :
00DA 528 ADD_PACKED:
59 57 90 00DA 529 MOV B R7,R9 ; Use sign of second string for output
03 59 E9 00DD 530 BL B C R9,10$ ; Check if sign is negative
5B 08 88 00E0 531 B I S B #PSL$M_N,R11 ; ... so the saved N-bit can be set
00E3 532 :
00E3 533 MARK POINT ADD SUB 24
56 61 0F 8B 00E3 534 10$: B I C B 3 #B00001111,(R1),R6 ; Get least significant digit to R6
00E7 535 MARK POINT ADD SUB 24
57 63 0F 8B 00E7 536 B I C B 3 #B00001111,(R3),R7 ; Get least significant digit to R7
58 D4 00EB 537 CL R L R8 ; Start the add with CARRY off
0075 30 00ED 538 B S B W ADD_PACKED_BYTE_R6_R7 ; Add the two low order digits
00F0 539 :
00F0 540 : The following set of instructions computes the number of bytes in the two
00F0 541 : strings and, if necessary, performs a switch so that R0 and R1 always
00F0 542 : describe the shorter of the two strings.
00F0 543 :
50 50 04 01 EF 00F0 544 EXT Z V #1,#4,R0,R0 ; Convert digit count to byte count
52 52 04 01 EF 00F5 545 EXT Z V #1,#4,R2,R2 ; Do it for both strings
52 50 D1 00FA 546 C M P L R0,R2 ; We want to compare the byte counts
09 1B 00FD 547 B L E Q U 20$ ; Skip the swap if we're already correct
56 50 7D 00FF 548 M O V Q R0,R6 ; Save the longer
50 52 7D 0102 549 M O V Q R2,R0 ; Store the shorter on R0 and R1
52 56 7D 0105 550 M O V Q R6,R2 ; ... and store the longer in R2 and R3
52 50 C2 0108 551 20$: S U B L R0,R2 ; Make R2 a difference (R2 GEQU 0)
010B 552 :
010B 553 : R0 now contains the number of bytes remaining in the shorter string.
010B 554 : R2 contains the difference in bytes between the two input strings.
010B 555 :
50 D5 010B 556 T S T L R0 ; Does shorter string have any room?
06 13 010D 557 B E Q L 40$ ; Skip loop if no room at all
010F 558 :
004D 30 010F 559 30$: B S B W ADD_PACKED_BYTE_STRING ; Add the next two bytes together
FA 50 F5 0112 560 S O B G T R R0,30$ ; Check for end of loop
0115 561 :
52 D5 0115 562 40$: T S T L R2 ; Does longer string have any room?
16 13 0117 563 B E Q L 70$ ; Skip next loops if all done
0119 564 :
0D 58 E9 0119 565 50$: B L B C R8,60$ ; Life is simple if CARRY clear
011C 566 :
56 D4 011C 567 CL R L R6 ; Otherwise, CARRY must propagate
011E 568 MARK POINT ADD SUB 24
57 73 9A 011E 569 M O V Z B L -(R3),R7 ; So add CARRY to single string
0041 30 0121 570 B S B W ADD_PACKED_BYTE_R6_R7 ; Use the special entry point
```

```
F2 52 F5 0124 571 SOBGTR R2,50$ ; Check for this string exhausted
06 11 0127 572 BRB 70$ ; Join common completion code
0129 573
0129 574 MARK_POINT ADD_SUB_24
75 73 90 0129 575 60$: MOVB -(R3),-(R5) ; Simply move src to dst if no CARRY
FA 52 F5 012C 576 SOBGTR R2,60$ ; ... until we're all done
012F 577
75 58 90 012F 578 70$: MOVB R8,-(R5) ; Store the final CARRY
0132 579
0132 580 ;+
0132 581 ; At this point, the result has been computed. That result must be moved to
0132 582 ; its ultimate destination, noting whether any nonzero digits are stored
0132 583 ; so that the Z-bit will have its correct setting.
0132 584
0132 585 Input Parameters:
0132 586
0132 587 R9<7:0> - Sign of result in preferred form
0132 588 R11<3:0> - Saved condition codes
0132 589 R11<31> - Indicates whether to set saved R4 to zero
0132 590
0132 591 (SP) - Saved R5, high address end of destination string
0132 592 ; -
0132 593
0132 594 ADD_SUBTRACT_EXIT:
55 6E 01 C1 0132 595 ADDL3 #1,(SP),R5 ; Point R5 beyond real destination
51 18 AE 9E 0136 596 MOVAB 24(SP),R1 ; R1 locates the saved result
010C 30 013A 597 BSBW STORE RESULT ; Store the result and record the Z-bit
12 5B 02 E0 013D 598 BBS #PSL$V_Z,R11,100$ ; Step out of line for minus zero check
0141 599
0141 600 MARK_POINT ADD SUB_24
9E 04 00 59 F0 0141 601 80$: INSV R9,#0,#4,a(SP)+ ; The sign can finally be stored
5E 14 C0 0146 602 ADDL #20,SP ; Get rid of intermediate buffer
03 5B 1F E1 0149 603 BBC #ADD_SUB_V_ZERO_R4,R11,90$ ; Branch if 4-operand opcode
10 AE D4 014D 604 CLRL 16(SP) ; Clear saved R4 to return zero
FEAD' 31 0150 605 90$: BRW VAX$DECIMAL_EXIT ; Exit through common code path
0153 606
0153 607 ; If the result is negative zero, then the N-bit is cleared and the sign
0153 608 ; is changed to a plus sign.
0153 609
0153 610 100$: BICB #PSL$M_N,R11 ; Clear the N-bit unconditionally
E7 5B 08 8A 0153 611 BBS #PSL$V_V,R11,80$ ; Do not change the sign on overflow
5B 01 E0 0156 612 MOVAB #12,R9 ; Make sure that the sign is plus
59 0C 90 015A 613 BRB 80$ ; ... and rejoin the exit code
E2 11 015D 613
```

```
015F 615 .SUBTITLE ADD_PACKED_BYTE - Add Two Bytes Containing Decimal Digits
015F 616
015F 617 Functional Description:
015F 618
015F 619 This routine adds together two bytes containing decimal digits and
015F 620 produces a byte containing the sum that is stored in the output
015F 621 string. Each of the input bytes is converted to a binary number
015F 622 (with a table-driven conversion), the two numbers are added, and
015F 623 the sum is converted back to two decimal digits stored in a byte.
015F 624
015F 625 This routine makes no provisions for bytes that contain illegal
015F 626 decimal digits. We are using the UNPREDICTABLE statement in the
015F 627 architectural description of the decimal instructions to its fullest.
015F 628
015F 629 The bytes that contain a pair of packed decimal digits can either
015F 630 exist in packed decimal strings located by R1 and R3 or they can
015F 631 be stored directly in registers. In the former case, the digits must
015F 632 be extracted from registers before they can be used in later operations
015F 633 because the sum will be used as an index register.
015F 634
015F 635 For entry at ADD_PACKED_BYTE_STRING:
015F 636
015F 637 Input Parameters:
015F 638
015F 639 R1 - Address one byte beyond first byte that is to be added
015F 640 R3 - Address one byte beyond second byte that is to be added
015F 641 R5 - Address one byte beyond location to store sum
015F 642
015F 643 R8 - Carry from previous byte (R8 is either 0 or 1)
015F 644
015F 645 Implicit Input:
015F 646
015F 647 R6 - Scratch
015F 648 R7 - Scratch
015F 649
015F 650 Output Parameters:
015F 651
015F 652 R1 - Decreased by one to point to current byte in first input string
015F 653 R3 - Decreased by one to point to current byte in second input string
015F 654 R5 - Decreased by one to point to current byte in output string
015F 655
015F 656 R8 - Either 0 or 1, reflecting whether this most recent ADD resulted
015F 657 in a CARRY to the next byte.
015F 658
015F 659 For entry at ADD_PACKED_BYTE_R6_R7:
015F 660
015F 661 Input Parameters:
015F 662
015F 663 R6 - First byte containing decimal digit pair
015F 664 R7 - Second byte containing decimal digit pair
015F 665
015F 666 R5 - Address one byte beyond location to store sum
015F 667
015F 668 R8 - Carry from previous byte (R8 is either 0 or 1)
015F 669
015F 670 Output Parameters:
015F 671
```

015F 672 : R5 - Decreased by one to point to current byte in output string
015F 673 :
015F 674 : R8 - Either 0 or 1, reflecting whether this most recent ADD resulted
015F 675 : in a CARRY to the next byte.
015F 676 :
015F 677 : Side Effects:
015F 678 : R6 and R7 are modified by this routine
015F 679 :
015F 680 : R0, R2, R4, and R9 (and, of course, R10 and R11) are preserved
015F 681 : by this routine
015F 682 :
015F 683 : Assumptions:
015F 684 : This routine makes two important assumptions.
015F 685 :
015F 686 : 1. If both of the input bytes contain only legal decimal digits, then
015F 687 : it is only necessary to subtract 100 at most once to put all
015F 688 : possible sums in the range 0..99. That is,
015F 689 :
015F 690 : $99 + 99 + 1 = 199$ LSS 200
015F 691 :
015F 692 : 2. The result will be checked in some way to determine whether the
015F 693 : result is nonzero so that the Z-bit can have its correct setting.
015F 694 :
015F 695 :
015F 696 :
015F 697 :
015F 698 : ADD_PACKED_BYTE_STRING:
015F 699 :
015F 700 : MARK POINT ADD_SUB_BSBW_24
56 71 9A 015F 701 : MOVZBL -(R1),R6 : Get byte from first string
0162 702 : MARK POINT ADD_SUB_BSBW_24
57 73 9A 0162 703 : MOVZBL -(R3),R7 : Get byte from second string
0165 704 :
0165 705 : VAX\$ADD_PACKED_BYTE_R6_R7:: : ASHP also uses this routine
0165 706 : ADD_PACKED_BYTE_R6_R7:-
56 0000'CF46 90 0165 707 : MOVB DECIMAL\$PACKED_TO_BINARY_TABLE[R6],-
0168 708 : R6 : Convert digits to binary
57 0000'CF47 90 0168 709 : MOVB DECIMAL\$PACKED_TO_BINARY_TABLE[R7],-
0171 710 : R7 : Convert digits to binary
57 56 80 0171 711 : ADDB R6,R7 : Form their sum
57 58 80 0174 712 : ADDB R8,R7 : Add CARRY from last step
58 58 94 0177 713 : CLRB R8 : Assume no CARRY this time
63 8F 57 91 0179 714 : CMPB R7,#99 : Check for CARRY
58 07 18 017D 715 : BLEQU 10\$: Branch if within bounds
57 58 01 90 017F 716 : MOVB #1,R8 : Propagate CARRY to next step
57 64 8F 82 0182 717 : SUBB #100,R7 : Put R7 into interval 0..99
75 0000'CF47 90 0186 718 : 10\$: MOVB DECIMAL\$BINARY_TO_PACKED_TABLE[R7],-
018C 719 : -(R5) : Store converted sum byte
05 018C 720 : RSB

```
0180 722 .SUBTITLE SUBTRACT_PACKED - Subtract Two Packed Decimal Strings
0180 723
0180 724
0180 725
0180 726
0180 727
0180 728
0180 729
0180 730
0180 731
0180 732
0180 733
0180 734
0180 735
0180 736
0180 737
0180 738
0180 739
0180 740
0180 741
0180 742
0180 743
0180 744
0180 745
0180 746
0180 747
0180 748
0180 749
0180 750
0180 751
0180 752
0180 753
0180 754
0180 755
0180 756
0180 757
0180 758
0180 759
0180 760
0180 761
0180 762
0180 763
0180 764
0180 765
0180 766
0180 767
0180 768
0180 769
0180 770
0180 771
0180 772
0180 773
0180 774
0180 775
0180 776
0180 777
0180 778
```

Functional Description:

This routine takes two packed decimal strings whose descriptors are passed as input parameters, subtracts one string from the other, and places their sum into another (perhaps identical) packed decimal string.

At the present time, the result is placed into a 16-byte storage area while the difference is being evaluated. This drastically reduces the number of different cases that must be dealt with as each pair of bytes in the two input strings is added.

The signs of the two input strings have already been dealt with so this routine performs subtraction in all cases, even if the original entry was at ADDP4 or ADDP6.

Input Parameters:

R0<4:0> - Number of digits in first input decimal string
R1 - Address of least significant digit of first input decimal string (the byte containing the sign)

R2<4:0> - Number of digits in second input decimal string
R3 - Address of least significant digit of second input decimal string (the byte containing the sign)

R4<4:0> - Number of digits in output decimal string
R5 - Address of one byte beyond least significant digit of intermediate string stored on the stack

R6<3:0> - Sign of first input string in preferred form
R7<3:0> - Sign of second input string in preferred form

R11 - Saved PSL (Z-bit is set, other condition codes are clear)

(SP) - Saved R5, address of least significant digit of ultimate destination string.
4(SP) - Beginning of 20-byte buffer to hold intermediate result

Output Parameters:

The particular input operation (ADDPx or SUBPx) is completed in this routine. See the routine headers for the four routines that request addition or subtraction for a list of output parameters from this routine.

Algorithm for Choice of Sign:

The choice of sign for the output string is not nearly so straightforward as it is in the case of addition. One approach that is often taken is to make a reasonable guess at the sign of the result. If the final subtraction causes a BORROW, then the choice was incorrect. The sign must be changed and the result must be replaced by its tens complement.

```
018D 779 : This routine does not guess. Instead, it chooses the input string of
018D 780 : the larger absolute magnitude as the minuend for this internal
018D 781 : routine and chooses its sign as the sign of the result.
018D 782 : This algorithm is actually more efficient than the reasonable
018D 783 : guess method and is probably better than a guess method that is never
018D 784 : wrong. All complete bytes that are processed in the sign evaluation
018D 785 : preprocessing loop are eliminated from consideration in the
018D 786 : subtraction loop, which has a higher cost per byte.
018D 787 :
018D 788 : The actual algorithm is as follows. (Note that both input strings have
018D 789 : already had leading zeros stripped so their lengths reflect
018D 790 : significant digits.)
018D 791 :
018D 792 : 1. If the two strings have unequal lengths, then choose the sign of
018D 793 : the string that has the longer length.
018D 794 :
018D 795 : 2. For strings of equal length, choose the sign of the string whose
018D 796 : most significant byte is larger in magnitude.
018D 797 :
018D 798 : 3. If the most significant bytes test equal, then decrease the
018D 799 : lengths of each string by one byte, drop the previous most
018D 800 : significant bytes, and go back to step 2.
018D 801 :
018D 802 : 4. If the two strings test equal, it is not necessary to do any
018D 803 : subtraction. The result is identically zero.
018D 804 :
018D 805 : Note that the key to this routine's efficiency is that high order
018D 806 : bytes that test equal in this loop are dropped from consideration in
018D 807 : the more complicated subtraction loop.
018D 808 :--
018D 809 :
018D 810 SUBTRACT_PACKED:
50 50 04 01 EF 018D 811 EXTZV #1,#4,R0,R0 : Convert digit count to byte count
52 52 04 01 EF 0192 812 EXTZV #1,#4,R2,R2 : Do it for both strings
52 50 D1 0197 813 CMPL R0,R2 : We want to compare the byte counts
3C 1F 019A 814 BLSSU 40$ : R0/R1 represent the smaller string
2A 1A 019C 815 BGTRU 30$ : R2/R3 represent the smaller string
019E 816 :
019E 817 : The two input strings have an equal number of bytes. Compare magnitudes to
019E 818 : determine which string is really larger. If the two strings test equal, then
019E 819 : skip the entire subtraction loop.
019E 820 :
58 51 50 C3 019E 821 SUBL3 R0,R1,R8 : Point R8 to low address end of R0/R1
59 53 52 C3 01A2 822 SUBL3 R2,R3,R9 : Point R9 to low address end of R2/R3
50 D5 01A6 823 TSTL R0 : See if both strings have zero bytes
0C 13 01A8 824 BEQL 20$ : Still need to check low order digit
01AA 825 :
01AA 826 MARK_POINT ADD_SUB_24
89 88 91 01AA 827 10$: CMPB (R8)+,(R9)+ : Compare most significant bytes
29 1F 01AD 828 BLSSU 40$ : R0/R1 represent the smaller string
17 1A 01AF 829 BGTRU 30$ : R2/R3 represent the smaller string
52 D7 01B1 830 DECL R2 : Keep R2 in step with R0
F4 50 F5 01B3 831 SOBGTR R0,10$ : ... which gets decremented here
01B6 832 :
01B6 833 : At this point, we have reduced both input strings to single bytes that
01B6 834 : contain a sign "digit" and may contain a digit in the high order nibble
01B6 835 : if the original digit counts were nonzero.
```

```

      01B6 836
      01B6 837
58 68 OF 88 01B6 838 20$: MARK_POINT ADD_SUB_24
      01BA 839 BICB3 #*B00001111,(R8),R8 : Look only at digit, ignoring sign
59 69 OF 88 01BA 840 MARK_POINT ADD_SUB_24
      59 58 91 01BE 841 BICB3 #*B00001111,(R9),R9 : Get the digit from the other string
      15 1F 01C1 842 CMPB R8,R9 : Compare these digits
      03 1A 01C3 843 BLSSU 40$ : R0/R1 represent the smaller string
      01C5 844 BGTRU 30$ : R2/R3 represent the smaller string
      01C5 845 : The two strings have identical magnitudes. Enter the end processing code
      01C5 846 : with the intermediate result unchanged (that is, zero).
      01C5 847
      FF6A 31 01C5 848 BRW ADD_SUBTRACT_EXIT : Join the common completion code
      01C8 849
      01C8 850 : The string described by R0 and R1 has the larger magnitude. Choose its sign.
      01C8 851 : Then swap the two string descriptors so that the main subtraction loops
      01C8 852 : always have R2 and R3 describing the larger string. Note that the use of
      01C8 853 : R6 and R7 as scratch leaves R7<31:8> in an UNPREDICTABLE state.
      01C8 854
      59 56 90 01C8 855 30$: MOVB R6,R9 : Load preferred sign into R9
      56 50 7D 01CB 856 MOVQ R0,R6 : Save the longer
      50 52 7D 01CE 857 MOVQ R2,R0 : Store the shorter on R0 and R1
      52 56 7D 01D1 858 MOVQ R6,R2 : ... and store the longer in R2 and R3
      57 D4 01D4 859 CLRL R7 : Insure that R7<31:8> is zero
      03 11 01D6 860 BRB 50$ : Continue along common code path
      01D8 861
      01D8 862 : The string described by R2 and R3 has the larger magnitude. Choose its sign.
      01D8 863
      59 57 90 01D8 864 40$: MOVB R7,R9 : Load preferred sign into R9
      01D8 865
      52 50 C2 01D8 866 50$: SUBL R0,R2 : Make R2 a difference (R2 GEQU 0)
      03 59 E9 01DE 867 BLBC R9,60$ : Check if sign is negative
      58 08 88 01E1 868 BISB #PSLSM_N,R11 : ... so the saved N-bit can be set
      01E4 869
      01E4 870 MARK_POINT ADD_SUB_24
      56 61 OF 88 01E4 871 60$: BICB3 #*B00001111,(R1),R6 : Get least significant digit to R6
      01E8 872 MARK_POINT ADD_SUB_24
      57 63 OF 88 01E8 873 BICB3 #*B00001111,(R3),R7 : Get least significant digit to R7
      58 D4 01EC 874 CLRL R8 : Start subtracting with BORROW off
      0032 30 01EE 875 BSBW SUB_PACKED_BYTE_R6_R7 : Subtract the two low order digits
      01F1 876
      01F1 877 : R0 contains the number of bytes remaining in the smaller string
      01F1 878 : R2 contains the difference in bytes between the two input strings
      01F1 879
      50 D5 01F1 880 TSTL R0 : Does smaller string have any room?
      06 13 01F3 881 BEQL 80$ : Skip loop if no room at all
      01F5 882
      0025 30 01F5 883 70$: BSBW SUB_PACKED_BYTE_STRING : Subtract the next two bytes
      FA 50 F5 01F8 884 SOBGTR R0,70$ : Check for end of loop
      01FB 885
      52 D5 01FB 886 80$: TSTL R2 : Does one of the strings have more?
      16 13 01FD 887 BEQL 110$ : Skip next loops if all done
      01FF 888
      OD 58 E9 01FF 889 90$: BLBC R8,100$ : Life is simple if BORROW clear
      56 D4 0202 890 CLRL R6 : Otherwise, BORROW must propagate
      0204 891 MARK_POINT ADD_SUB_24
      0204 892
```

```
57 73 9A 0204 893 MOVZBL -(R3),R7 ; So subtract BORROW from single string
    0019 30 0207 894 BSBW SUB_PACKED_BYTE_R6_R7 ; Use the special entry point
    F2 52 F5 020A 895 SOBGTR R2,90$ ; Check for this string exhausted
    06 11 020D 896 BRB 110$ ; Join common completion code
    020F 897
    020F 898
75 73 90 020F 899 100$: MARK_POINT ADD_SUB_24 ; Simply move src to dst if no BORROW
    FA 52 F5 0212 900 SOBGTR R2,100$ ; ... until we're all done
    0215 901
    0215 902 110$:
    0215 903
    0215 904 : : : ***** BEGIN TEMP *****
    0215 905 : : :
    0215 906 : : : THE FOLLOWING HALT INSTRUCTION SHOULD BE REPLACED WITH THE CORRECT
    0215 907 : : : ABORT CODE.
    0215 908 : : :
    0215 909 : : : THE HALT IS SIMILAR TO THE
    0215 910 : : :
    0215 911 : : : MICROCODE CANNOT GET HERE
    0215 912 : : :
    0215 913 : : : ERRORS THAT OTHER IMPLEMENTATIONS USE.
    0215 914 : : :
    58 D5 0215 915 tstl r8 ; If BORROW is set here, we blew it
    01 13 0217 916 beql 120$ ; Branch out if OK
    00 00 0219 917 halt ; This will cause an OPCDEC exception
    021A 918 120$:
    021A 919 : : :
    021A 920 : : : ***** END TEMP *****
    021A 921
FF15 31 021A 922 BRW ADD_SUBTRACT_EXIT ; Join common completion code
```

```
0210 924 .SUBTITLE SUB_PACKED_BYTE - Subtract Two Bytes Containing Decimal Digi
0210 925
0210 926 :+ Functional Description:
0210 927
0210 928 This routine takes as input two bytes containing decimal digits and
0210 929 produces a byte containing their difference. This result is stored in
0210 930 the output string. Each of the input bytes is converted to a binary
0210 931 number (with a table-driven conversion), the first number is
0210 932 subtracted from the second, and the difference is converted back to
0210 933 two decimal digits stored in a byte.
0210 934
0210 935 This routine makes no provisions for bytes that contain illegal
0210 936 decimal digits. We are using the UNPREDICTABLE statement in the
0210 937 architectural description of the decimal instructions to its fullest.
0210 938
0210 939 The bytes that contain a pair of packed decimal digits can either
0210 940 exist in packed decimal strings located by R1 and R3 or they can
0210 941 be stored directly in registers. In the former case, the digits must
0210 942 be extracted from registers before they can be used in later operations
0210 943 because the difference will be used as an index register.
0210 944
0210 945 For entry at SUB_PACKED_BYTE_STRING:
0210 946
0210 947 Input Parameters:
0210 948
0210 949 R1 - Address one byte beyond byte containing subtrahend
0210 950 R3 - Address one byte beyond byte containing minuend
0210 951 R5 - Address one byte beyond location to store difference
0210 952
0210 953 R8 - BORROW from previous byte (R8 is either 0 or 1)
0210 954
0210 955 Implicit Input:
0210 956
0210 957 R6 - Scratch
0210 958 R7 - Scratch
0210 959
0210 960 Output Parameters:
0210 961
0210 962 R1 - Decreased by one to point to current byte
0210 963 in subtrahend string
0210 964 R3 - Decreased by one to point to current byte
0210 965 in minuend string
0210 966 R5 - Decreased by one to point to current byte
0210 967 in difference string
0210 968
0210 969 R8 - Either 0 or 1, reflecting whether this most recent
0210 970 subtraction resulted in a BORROW from the next byte.
0210 971
0210 972 For entry at SUB_PACKED_BYTE_R6_R7:
0210 973
0210 974 Input Parameters:
0210 975
0210 976 R6<7:0> - Byte containing decimal digit pair for subtrahend
0210 977 R6<31:8> - MBZ
0210 978 R7<7:0> - Byte containing decimal digit pair for minuend
0210 979 R7<31:8> - MBZ
0210 980
```

```
021D 981 : R5 - Address one byte beyond location to store difference
021D 982 :
021D 983 : R8 - BORROW from subtraction of previous byte
021D 984 : (R8 is either 0 or 1)
021D 985 :
021D 986 : Output Parameters:
021D 987 :
021D 988 : R5 - Decreased by one to point to current byte
021D 989 : in difference string
021D 990 :
021D 991 : R8 - Either 0 or 1, reflecting whether this most recent
021D 992 : subtraction resulted in a BORROW from the next byte.
021D 993 :
021D 994 : Side Effects:
021D 995 :
021D 996 : R6 and R7 are modified by this routine
021D 997 :
021D 998 : R0, R2, R4, and R9 (and, of course, R10 and R11) are preserved
021D 999 : by this routine
021D 1000 :
021D 1001 : Assumptions:
021D 1002 :
021D 1003 : This routine makes two important assumptions.
021D 1004 :
021D 1005 : 1. If both of the input bytes contain only legal decimal digits, then
021D 1006 : it is only necessary to add 100 at most once to put all
021D 1007 : possible differences in the range 0..99. That is,
021D 1008 :
021D 1009 : 0 - 99 - 1 = -100
021D 1010 :
021D 1011 : 2. The result will be checked in some way to determine whether the
021D 1012 : result is nonzero so that the Z-bit can have its correct setting.
021D 1013 :
021D 1014 :
021D 1015 : SUB_PACKED_BYTE_STRING:
021D 1016 :
021D 1017 : MARK POINT ADD_SUB_BSBW_24
56 71 9A 021D 1018 : MOVZBL -(R1),R6 : Get byte from first string
57 73 9A 0220 1019 : MARK POINT ADD_SUB_BSBW_24
0220 1020 : MOVZBL -(R3),R7 : Get byte from second string
0223 1021 :
0223 1022 : SUB_PACKED_BYTE_R6_R7:
56 0000'CF46 90 0223 1023 : MOVBL DECIMAL$PACKED_TO_BINARY_TABLE[R6],-
0229 1024 : R6 : Convert digits to binary
57 0000'CF47 90 0229 1025 : MOVBL DECIMAL$PACKED_TO_BINARY_TABLE[R7],-
022F 1026 : R7 : Convert digits to binary
57 56 82 022F 1027 : SUBB R6,R7 : Form their difference
57 58 82 0232 1028 : SUBB R8,R7 : Include BORROW from last step
04 19 0235 1029 : BLSS 10$ : Branch if need to BORROW
58 94 0237 1030 : CLRB R8 : No BORROW next time
07 11 0239 1031 : BRB 20$ : Join common exit code
023B 1032 :
57 64 8F 80 023B 1033 10$: ADDB #100,R7 : Put R7 into interval 0..99
58 01 90 023F 1034 : MOVBL #1,R8 : Propagate BORROW to next step
0242 1035 :
75 0000'CF47 90 0242 1036 20$: MOVBL DECIMAL$BINARY_TO_PACKED_TABLE[R7],-
024B 1037 : -(R5) : Store converted sum byte
```

VAXDECIMAL_ARITHMETIC
V04-000

K 10
- VAX-11 Packed Decimal Arithmetic Instr 16-SEP-1984 01:33:44 VAX/VMS Macro V04-00 Page 23
SUB_PACKED_BYTE - Subtract Two Bytes Con 5-SEP-1984 00:44:34 [EMULAT.SRC]VAXARITH.MAR;1 (12)
05 0248 1038 RSB

```
0249 1040 .SUBTITLE STORE_RESULT - Store Decimal String
0249 1041
0249 1042 Functional Description:
0249 1043
0249 1044 This routine takes a packed decimal string that typically contains
0249 1045 the result of an arithmetic operation and stores it in another
0249 1046 decimal string whose descriptor is specified as an input parameter
0249 1047 to the original arithmetic operation.
0249 1048
0249 1049 The string is stored from the high address end (least significant
0249 1050 digits) to the low address end (most significant digits). This order
0249 1051 allows all of the special cases to be handled in the simplest fashion.
0249 1052
0249 1053 Input Parameters:
0249 1054
0249 1055 R1 - Address one byte beyond high address end of input string
0249 1056 (Note that this string must be at least 17 bytes long.)
0249 1057
0249 1058 R4<4:0> - Number of digits in ultimate destination
0249 1059 R5 - Address one byte beyond destination string
0249 1060
0249 1061 R11 - Contains saved condition codes
0249 1062
0249 1063 Implicit Input:
0249 1064
0249 1065 The input string must be at least 17 bytes long to contain a potential
0249 1066 carry out of the highest digit when doing an add of two large numbers.
0249 1067 This carry out of the last byte will be detected and reported as a
0249 1068 decimal overflow, either as an exception or simply by setting the V-bit.
0249 1069
0249 1070 The least significant digit (highest addressed byte) cannot contain a
0249 1071 sign digit because that would cause the Z-bit to be incorrectly cleared.
0249 1072
0249 1073 Output Parameters:
0249 1074
0249 1075 R11<PSLSV_Z> - Cleared if a nonzero digit is stored in output string
0249 1076 R11<PSLSV_V> - Set if a nonzero digit is detected after the output
0249 1077 string is exhausted
0249 1078
0249 1079 A portion of the result (dictated by the size of R4 on input) is
0249 1080 moved to the destination string.
0249 1081
0249 1082
0249 1083 STORE_RESULT:
0249 1084 INCL R4 ; Want number of "complete" bytes in
0249 1085 ASHL #-1,R4,R0 ; output string
0249 1086 BEQL 30$ ; Skip first loop if none
0249 1087
0249 1088 MARK_POINT ADD_SUB_BSBW_24
0249 1089 10$: MOVB -(R1),-(R5) ; Move the next complete byte
0249 1090 BEQL 20$ ; Check whether to clear Z-bit
0249 1091 BICB #PSLSM_Z,R11 ; Clear Z-bit if nonzero
0249 1092 20$: SOBGTR R0,10$ ; Keep going?
0249 1093
0249 1094 30$: BLBC R4,50$ ; Was original R4 odd? Branch if yes
0249 1095 MARK_POINT ADD_SUB_BSBW_24
0249 1096 BICB3 #B11110000,-(R1),-(R5) ; If R4 was even, store half a byte
```

50	54	FF	8F	0B	D6	78	13
					75	71	90
						03	13
					5B	04	8A
					F5	50	F5
						10	54
						E9	
75	71	F0	8F	8B			

```

      03 13 0265 1097      BEQL 40$      ; Need to check for zero here, too
5B 04 8A 0267 1098      BICB #PSLSM_Z,R11 ; Clear Z-bit if nonzero
      026A 1099      MARK_POINT ADD_SUB_BSBW_24
61 F0 8F 93 026A 1100 40$: BITB #B11110000,(R1) ; If high order nibble is nonzero,
      13 12 026E 1101      BNEQ 70$      ; ... then overflow has occurred
      0270 1102
      0270 1103 ; The entire destination has been stored. We must now check whether any of
      0270 1104 ; the remaining input string is nonzero and set the V-bit if nonzero is
      0270 1105 ; detected. Note that at least one byte of the output string has been examined
      0270 1106 ; in all cases already. This makes the next byte count calculation correct.
      0270 1107
      0270 1108 50$: DECL R4 ; Restore R4 to its original self
50 54 04 54 D7 0270 1109      EXTZV #1,#4,R4,R0 ; Extract a byte count
      01 EF 0272 1110      SUBB3 R0,#16,R0 ; Loop count is 16 minus byte count
50 50 10 50 83 0277 1111
      027B 1112 ; Note that the loop count can never be zero because we are testing a 17-byte
      027B 1113 ; string and the largest output string can be 16 bytes long.
      027B 1114
      027B 1115      MARK_POINT ADD_SUB_BSBW_24
      71 95 027B 1116 60$: TSTB -(R1) ; Check next byte for nonzero
      04 12 027D 1117      BNEQ 70$      ; Nonzero means overflow has occurred
F9 50 F5 027F 1118      SOBGTR R0,60$ ; Check for end of this loop
      0282 1119
      05 0282 1120      RSB ; This is return path for no overflow
      0283 1121
5B 02 88 0283 1122 70$: BISB #PSLSM_V,R11 ; Indicate that overflow has occurred
      05 0286 1123      RSB ; ... and return to the caller
```

```

0287 1125 .SUBTITLE VAX$MULP - Multiply Packed
0287 1126
0287 1127 :+ Functional Description:
0287 1128
0287 1129 The multiplicand string specified by the multiplicand length and
0287 1130 multiplicand address operands is multiplied by the multiplier string
0287 1131 specified by the multiplier length and multiplier address operands. The
0287 1132 product string specified by the product length and product address
0287 1133 operands is replaced by the result.
0287 1134
0287 1135 Input Parameters:
0287 1136
0287 1137 R0 - mulrlen.rw Number of digits in multiplier string
0287 1138 R1 - mulraddr.ab Address of multiplier string
0287 1139 R2 - mulrlen.rw Number of digits in multiplicand string
0287 1140 R3 - muldaddr.ab Address of multiplicand string
0287 1141 R4 - prodlen.rw Number of digits in product string
0287 1142 R5 - prodaddr.ab Address of product string
0287 1143
0287 1144 Output Parameters:
0287 1145
0287 1146 R0 = 0
0287 1147 R1 = Address of the byte containing the most significant digit of
0287 1148 the multiplier string
0287 1149 R2 = 0
0287 1150 R3 = Address of the byte containing the most significant digit of
0287 1151 the multiplicand string
0287 1152 R4 = 0
0287 1153 R5 = Address of the byte containing the most significant digit of
0287 1154 the string containing the product
0287 1155
0287 1156 Condition Codes:
0287 1157
0287 1158 N <- product string LSS 0
0287 1159 Z <- product string EQL 0
0287 1160 V <- decimal overflow
0287 1161 C <- 0
0287 1162
0287 1163 Register Usage:
0287 1164
0287 1165 This routine uses all of the general registers. The condition codes
0287 1166 are computed at the end of the instruction as the final result is
0287 1167 stored in the product string. R11 is used to record the condition
0287 1168 codes.
0287 1169
0287 1170 Notes:
0287 1171
0287 1172 1. This routine uses a large amount of stack space to allow storage of
0287 1173 intermediate results in a convenient form. Specifically, each digit
0287 1174 pair of the longer input string is stored in binary in a longword on
0287 1175 the stack. In addition, 32 longwords are set aside to hold the product
0287 1176 intermediate result. Each longword contains a binary number between 0
0287 1177 and 99.
0287 1178
0287 1179 After the multiplication is complete, Each longword is removed from
0287 1180 the stack, converted to a packed decimal pair, and stored in the
0287 1181 output string. Any nonzero cells remaining on the stack after the

```

0287 1182 : output string has been completely filled are the indication of decimal
0287 1183 : overflow.
0287 1184 :
0287 1185 : The purpose of this method of storage is to avoid decimal/binary or
0287 1186 : even byte/longword conversions during the calculation of intermediate
0287 1187 : results.
0287 1188 :
0287 1189 : 2. Trailing zeros are removed from the larger string. All zeros in
0287 1190 : the shorter string are eliminated in the sense that no arithmetic
0287 1191 : is performed. The output array pointer is simply advanced to point
0287 1192 : to the next higher array element.
0287 1193 :
0287 1194 :
0287 1195 :
OFFF 8F BB 0287 1196 VAX\$MULP::
0288 1197 PUSHF #M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11> ; Save the lot
0288 1198 ESTABLISH HANDLER - ; Store address of access
0288 1199 ARITH_ACCVIO ; violation handler
0290 1200
0290 1201 ROPRAND_CHECK R4 ; Insure that R4 is LEQU 31
0298 1202
0298 1203 ROPRAND_CHECK R2 ; Insure that R2 is LEQU 31
02A3 1204 MARK_POINT MULP BSBW 0
FD5A' 30 02A3 1205 BSBW- DECIMAL\$STRIP_ZEROS_R2_R3 ; Strip high order zeros from R2/R3
02A6 1206
02A6 1207 ROPRAND_CHECK R0 ; Insure that R0 is LEQU 31
02AE 1208 MARK_POINT MULP BSBW 0
FD4F' 30 02AE 1209 BSBW- DECIMAL\$STRIP_ZEROS_R0_R1 ; Strip high order zeros from R0/R1
02B1 1210
50 50 04 01 EF 02B1 1211 EXTZV #1,#4,R0,R0 ; Convert digit count to byte count
50 50 D6 02B6 1212 INCL R0 ; Include least significant digit
02B8 1213
52 52 04 01 EF 02B8 1214 EXTZV #1,#4,R2,R2 ; Convert digit count to byte count
52 52 D6 02B0 1215 INCL R2 ; Include least significant digit
02BF 1216
52 50 D1 02BF 1217 CMPL R0,R2 ; See which string is larger
58 08 1A 02C2 1218 BGTRU 3\$; R2/R3 describes the longer string
58 52 7D 02C4 1219 MOVQ R2,R8 ; R8 and R9 describe the longer string
7E 50 7D 02C7 1220 MOVQ R0,-(SP) ; Shorter string descriptor also saved
06 11 02CA 1221 BRB 6\$
02CC 1222
58 50 7D 02CC 1223 3\$: MOVQ R0,R8 ; R8 and R9 describe the longer string
7E 52 7D 02CF 1224 MOVQ R2,-(SP) ; Shorter string descriptor also saved
02D2 1225
02D2 1226 ; Create space for the output array on the stack (32 longwords of zeros)
02D2 1227
50 08 D0 02D2 1228 6\$: MOVL #8,R0 ; Eight pairs of quadwords
02D5 1229
7E 7C 02D5 1230 10\$: CLRQ -(SP) ; Clear one pair
7E 7C 02D7 1231 CLRQ -(SP) ; ... and another
F9 50 F5 02D9 1232 SOBGTR R0,10\$; Do all eight pairs
02DC 1233
57 5E D0 02DC 1234 MOVL SP,R7 ; Store beginning of output array in R7
02DF 1235
02DF 1236 ; The longer input array will be stored on the stack as an array of
02DF 1237 ; longwords. Each array element contains a number between 0 and 99,
02DF 1238 ; representing a pair of digits in the original packed decimal string.

```
02DF 1239 : Because the units digit is stored with the sign in packed decimal format,
02DF 1240 : it is necessary to shift the number as we store it. This is accomplished by
02DF 1241 : multiplying the number by ten.
02DF 1242 :
02DF 1243 : The longer array is described by R8 (byte count) and R9 (address of most
02DF 1244 : significant digit pair).
02DF 1245 :
55 58 59 C1 02DF 1246 ADDL3 R9,R8,R5 ; Point R5 beyond sign digit
54 58 D0 02E3 1247 MOVL R8,R4 ; R4 contains the loop count
02E6 1248 :
02E6 1249 : An array of longwords is allocated on the stack. R3 starts out pointing
02E6 1250 : at the longword beyond the top of the stack. The first remainder, guaranteed
02E6 1251 : to be zero, is "stored" here. The rest of the digit pairs are stored safely
02E6 1252 : below the top of the stack.
02E6 1253 :
53 58 CE 02E6 1254 MNEGL R8,R3 ; Stack grows toward lower addresses
5E 6E43 DE 02E9 1255 MOVAL (SP)[R3],SP ; Allocate the space
53 5E 04 C3 02ED 1256 SUBL3 #4,SP,R3 ; Point R3 at next lower longword
02F1 1257 :
02F1 1258 :
51 51 75 9A 02F1 1259 20$: MARK POINT MULP_R8
0000'CF41 9A 02F4 1260 MOVZBL -(R5),R1 ; Get next digit pair
02FA 1261 MOVZBL DECIMAL$PACKED_TO_BINARY_TABLE[R1],- ; Convert digits to binary
83 52 50 52 51 0A 7A 02FA 1262 EMUL #10,R1,R2,R0 ; Multiply by 10
00000064 8F 7B 02FF 1263 EDIV #100,R0,R2,(R3)+ ; Divide by 100
E6 54 F5 0308 1264 SOBGTR R4,20$
030B 1265 :
63 52 D0 030B 1266 MOVL R2,(R3) ; Store final quotient
59 5E D0 030E 1267 MOVL SP,R9 ; Remember array address in R9
6E48 DF 0311 1268 PUSHAL (SP)[R8] ; Store start of fixed size area
0314 1269 :
0314 1270 : Check for trailing zeros in the input array stored on the stack. If any are
0314 1271 : present, they are removed and the product array is adjusted accordingly.
0314 1272 :
89 D5 0314 1273 30$: TSTL (R9)+ ; Is next number zero?
08 12 0316 1274 BNEQ 40$ ; Leave loop if nonzero
57 04 C0 0318 1275 ADDL #4,R7 ; Advance output pointer to next element
F6 58 F5 031B 1276 SOBGTR R8,30$ ; Keep going
031E 1277 :
031E 1278 : If we drop through the loop, then the entire input array is zero. There is
031E 1279 : no need to perform any arithmetic because the product will be zero (and the
031E 1280 : output array on the stack starts out as zero). The only remaining work is
031E 1281 : to store the result in the output string and set the condition codes.
031E 1282 :
2C 11 031E 1283 BRB 70$ ; Exit to end processing
0320 1284 :
0320 1285 : Now multiply the input array by each successive digit pair. In order to
0320 1286 : allow R10 to continue to locate ARITH ACCVIO while we execute this loop, it
0320 1287 : is necessary to perform a small amount of register juggling. In essence,
0320 1288 : R8 and R9 switch the identity of the string that they describe.
0320 1289 :
59 04 C2 0320 1290 40$: SUBL #4,R9 ; Readjust input array pointer
7E 58 7D 0323 1291 MOVQ R8,-(SP) ; Save R8/R9 descriptor on stack
58 08 AE D0 0326 1292 MOVL 8(SP),R8 ; Point R8 at start of 32-longword array
0080 C8 7D 032A 1293 MOVQ <32*4>(R8),R8 ; Get descriptor that follows that array
59 58 C0 032F 1294 ADDL2 R8,R9 ; Point R9 beyond sign byte
0332 1295 :
```

```
53 87 DE 0332 1296 50$: MOVAL (R7)+,R3 ; Output array address to R3
0335 1297 MARK POINT MULP_AT_SP
51 79 9A 0335 1298 MOVZBL -(R9),R1 ; Next digit pair to R1
56 0000'CF41 9A 0338 1299 MOVZBL DECIMAL$PACKED_TO_BINARY_TABLE[R1],-
033E 1300 R6 ; Convert digits to binary
06 13 033E 1301 BEQL 60$ ; Skip the work if zero
54 6E 7D 0340 1302 MOVQ (SP),R4 ; Input array descriptor to R4/R5
0104 30 0343 1303 BSBW EXTEND_STRING_MULTIPLY ; Do the work
E9 58 F5 0346 1304 60$: SOBGTR R8,50$ ; Any more multiplier digits?
0349 1305
SE 08 C0 0349 1306 ADDL #8,SP ; Discard saved long string descriptor
034C 1307
SE 6E D0 034C 1308 70$: MOVL (SP),SP ; Remove input array from stack
034F 1309
034F 1310 ; At this point, the product string is located in a 32-longword array on
034F 1311 ; the top of the stack. Each longword corresponds to a pair of digits in
034F 1312 ; the output string. As digits are removed from the stack, they are checked
034F 1313 ; for nonzero to obtain the correct setting of the Z-bit. After the output
034F 1314 ; string has been filled, the remainder of the product string is removed from
034F 1315 ; the stack. If a nonzero result is detected at this stage, the V-bit is set.
034F 1316
54 59 20 D0 034F 1317 MOVL #32,R9 ; Set up array counter
0098 CE 7D 0352 1318 MOVQ < <32*4> + - ; Skip over 32-longword array
0357 1319 <2*4> + - ; and saved string descriptor
0357 1320 <4*4> >(SP),R4 ; to retrieve original R4 and R5
```

```
0357 1322 .SUBTITLE Common Exit Path for VAX$MULP and VAX$DIVP
0357 1323
0357 1324 :+ The code for VAX$MULP and VAX$DIVP merges at this point. The result is stored
0357 1325 : in an array of longwords at the top of the stack. The size of this array is
0357 1326 : stored in R9. The original R4 and R5 have been retrieved from the stack.
0357 1327
0357 1328 : Input Parameters:
0357 1329 :
0357 1330 : R4 - Contains byte count of destination string in R4 <1:4>
0357 1331 : R5 - Address of most significant digit of destination string
0357 1332 : R9 - Count of longwords in result array on stack
0357 1333 :
0357 1334 : Contents of result array
0357 1335 :
0357 1336 : Implicit Input:
0357 1337 :
0357 1338 : Signs of two input factors (multiplier and multiplicand or
0357 1339 : divisor and dividend)
0357 1340 :-
0357 1341
0357 1342 MULTIPLY_DIVIDE_EXIT:
0357 1343 MOVPSL R11 : Get current PSL
0357 1344 INSV #PSL$M_Z,#0,#4,R11 : Clear all codes except Z-bit
0357 1345 ESTABLISH HANDLER : Store address of access
0357 1346 : violation handler again
0357 1347 EXTZV #1,#4,R4,R3 : Excess byte count to R3
0357 1348 BEQL 125$ : Skip to single digit code
0357 1349 ADDL3 R3,R5,R7 : Remember address of sign byte
0357 1350 ADDL3 #1,R7,R5 : Point R5 beyond end of product string
0357 1351
0357 1352 80$: MOVL (SP)+,R1 : Remove next value from stack
0357 1353 BEQL 90$ : Do not clear Z-bit if zero
0357 1354 BICB2 #PSL$M_Z,R11 : Clear Z-bit
0357 1355
0357 1356 MARK_POINT MULP_DIVP_R9
0357 1357 90$: MOVB DECIMAL$BINARY_TO_PACKED_TABLE[R1],- : Store converted sum byte
0357 1358 : -(R5) : One less element on the stack
0357 1359 DECL R9 : Exit loop if result array exhausted
0357 1360 BLEQ 116$ : Keep going?
0357 1361 SOBGTR R3,80$
0357 1362
0357 1363 100$: BLBC R4,120$ : Different for even digit count
0357 1364
0357 1365 : The output string consists of an odd number of digits. A complete digit
0357 1366 : pair can be stored in the most significant (lowest addressed) byte of
0357 1367 : the product string.
0357 1368
0357 1369 MOVL (SP)+,R1 : Remove next value from stack
0357 1370 BEQL 110$ : Do not clear Z-bit if zero
0357 1371 BICB2 #PSL$M_Z,R11 : Clear Z-bit
0357 1372
0357 1373 MARK_POINT MULP_DIVP_R9
0357 1374 110$: MOVB DECIMAL$BINARY_TO_PACKED_TABLE[R1],- : Store converted sum byte
0357 1375 : -(R5) : One less element on the stack
0357 1376 DECL R9 : Exit loop if result array exhausted
0357 1377 BLEQ 116$ : Perform overflow check
0357 1378 BRB 140$
```

5B	04	00	5B	DC	0357	1343	MOVPSL	R11	:	Get current PSL
			5B	FO	0357	1344	INSV	#PSL\$M_Z,#0,#4,R11	:	Clear all codes except Z-bit
					0357	1345	ESTABLISH	HANDLER	:	Store address of access
53	54	04	01	EF	0357	1346		ARITH_ACCVIO	:	violation handler again
			3B	13	0357	1347	EXTZV	#1,#4,R4,R3	:	Excess byte count to R3
	57	55	53	C1	0357	1348	BEQL	125\$:	Skip to single digit code
	55	57	01	C1	0357	1349	ADDL3	R3,R5,R7	:	Remember address of sign byte
					0357	1350	ADDL3	#1,R7,R5	:	Point R5 beyond end of product string
		51	8E	D0	0357	1351				
			03	13	0357	1352	80\$: MOVL	(SP)+,R1	:	Remove next value from stack
		5B	04	8A	0357	1353	BEQL	90\$:	Do not clear Z-bit if zero
					0357	1354	BICB2	#PSL\$M_Z,R11	:	Clear Z-bit
					0357	1355				
75	0000'	CF41	90		0357	1356	MARK_POINT	MULP_DIVP_R9		
					0357	1357	90\$: MOVB	DECIMAL\$BINARY_TO_PACKED_TABLE[R1],-	:	Store converted sum byte
					0357	1358		-(R5)	:	One less element on the stack
		59	D7		0357	1359	DECL	R9	:	Exit loop if result array exhausted
		1C	15		0357	1360	BLEQ	116\$:	Keep going?
	EB	53	F5		0357	1361	SOBGTR	R3,80\$		
					0357	1362				
	22	54	E9		0357	1363	100\$: BLBC	R4,120\$:	Different for even digit count
					0357	1364				
					0357	1365			:	The output string consists of an odd number of digits. A complete digit
					0357	1366			:	pair can be stored in the most significant (lowest addressed) byte of
					0357	1367			:	the product string.
					0357	1368				
	51	8E	D0		0357	1369	MOVL	(SP)+,R1	:	Remove next value from stack
		03	13		0357	1370	BEQL	110\$:	Do not clear Z-bit if zero
	5B	04	8A		0357	1371	BICB2	#PSL\$M_Z,R11	:	Clear Z-bit
					0357	1372				
					0357	1373	MARK_POINT	MULP_DIVP_R9		
75	0000'	CF41	90		0357	1374	110\$: MOVB	DECIMAL\$BINARY_TO_PACKED_TABLE[R1],-	:	Store converted sum byte
					0357	1375		-(R5)	:	One less element on the stack
		59	D7		0357	1376	DECL	R9	:	Exit loop if result array exhausted
		04	15		0357	1377	BLEQ	116\$:	Perform overflow check
		3B	11		0357	1378	BRB	140\$		

```
039E 1379
039E 1380 ; This loop executes if the result array has fewer elements than the output
039E 1381 ; string. The remaining bytes in the output string are filled with zeros.
039E 1382 ; There is no need for an overflow check.
039E 1383
039E 1384 MARK_POINT MULP_DIVP_8
FB 75 94 039E 1385 114$: CLRB -(R5) ; Store another zero byte
53 F4 03A0 1386 116$: SOBGEQ R3,114$ ; Any more room in output string
38 11 03A3 1387
03A3 1388 BRB 150$ ; Determine sign of result
03A5 1389
03A5 1390 ; This code path is used in the case where the output digit count is 0 or 1.
03A5 1391 ; R5 must be advanced
03A5 1392
57 55 D0 03A5 1393 125$: MOVL R5,R7 ; Remember address of output sign byte
55 D6 03A8 1394 INCL R5 ; Advance R5 so common code can be used
DB 11 03AA 1395 BRB 100$ ; Join common code path
03AC 1396
03AC 1397 ; The output string consists of an even number of digits. Only the low order
03AC 1398 ; nibble is stored in the most significant (lowest addresses) byte. A zero is
03AC 1399 ; stored in the high order nibble. If the high order digit would have been
03AC 1400 ; nonzero, the V-bit is set and the overflow check is bypassed because there
03AC 1401 ; are faster ways to clean the stack if we do not have to check for nonzero
03AC 1402 ; at the same time.
03AC 1403
51 51 8E D0 03AC 1404 120$: MOVL (SP)+,R1 ; Remove next value from stack
51 0000'CF41 90 03AF 1405 MOVB DECIMAL$BINARY_TO_PACKED_TABLE[R1],-
03B5 1406 R1 ; Obtain converted sum byte
03B5 1407 MARK_POINT MULP_DIVP_R9
75 51 F0 8F 8B 03B5 1408 BICB3 #^XF0,R1,-(R5) ; Store byte, clearing high order nibble
03 13 03BA 1409 BEQL 130$ ; Do not clear Z-bit if zero
5B 04 8A 03BC 1410 BICB2 #PSLSM_2,R11 ; Clear Z-bit
51 F0 8F 93 03BF 1411 130$: BITB #^XF0,R1 ; Is high order nibble nonzero?
06 12 03C3 1412 BNEQ 133$ ; Yes, go set overflow bit
59 D7 03C5 1413 DECL R9 ; One less element on the stack
D7 15 03C7 1414 BLEQ 116$ ; Exit loop if result array exhausted
0B 11 03C9 1415 BRB 140$ ; Check rest of result array for nonzero
03CB 1416
03CB 1417 ; If we detect overflow, we need to adjust R9 to reflect the nonzero longword
03CB 1418 ; removed from the stack before we enter the next code block that sets the
03CB 1419 ; V-bit and cleans off the stack based on the contents of R9.
03CB 1420
59 D7 03CB 1421 133$: DECL R9 ; One more longword removed from stack
03CD 1422
03CD 1423 ; A nonzero digit has been discovered in a position that cannot be stored in
03CD 1424 ; the output string. Set the V-bit, remove the rest of the product array from
03CD 1425 ; the stack, and join the exit processing in the code that determines the sign
03CD 1426 ; of the product.
03CD 1427
5B 02 88 03CD 1428 135$: BISB #PSLSM_V,R11 ; Set the overflow bit
SE 6E49 DE 03D0 1429 MOVAL (SP)[R9],SP ; Clean off remaining product string
07 11 03D4 1430 BRB 150$ ; Go to code that determines the sign
03D6 1431
03D6 1432 ; The remainder of the product array must be removed from the stack. A nonzero
03D6 1433 ; result causes the V-bit to be set and the rest of the loop to be skipped.
03D6 1434 ; Note that there is always a nonzero loop count remaining at this point.
03D6 1435
```

```

      8E D5 03D6 1436 140$: TSTL (SP)+ ; Is next longword zero?
      F1 12 03D8 1437      BNEQ 133$ ; No, leave loop
      F9 59 F5 03DA 1438      SOBGTR R9,140$
      03DD 1439
      03DD 1440 ; The final product string has been stored and the V- and Z-bits have their
      03DD 1441 ; correct settings. The sign of the product must be determined from the
      03DD 1442 ; signs of the two input strings. Opposite signs produce a negative product.
      03DD 1443 ; Same signs (in any representation) produce a plus sign in the output string.
      03DD 1444
      SE 08 C0 03CD 1445 150$: ADDL #8,SP ; Discard saved string descriptor
      56 0C D0 03E0 1446      MOVL #12,R6 ; Assume final result is positive
      50 04 6E 03E3 1447      MOVQ (SP),R0 ; Retrieve original R0/R1 pair
      51 01 EF 03E6 1448      EXTZV #1,#4,R0,R0 ; Get byte count for first input string
      51 50 C0 03EB 1449      ADDL R0,R1 ; Point R1 to byte containing sign
      50 61 F0 8F 8B 03EE 1450      MARK_POINT MULP_DIVP_0
      03EE 1451      BICB3 #*B11110000,(R1),R0 ; R0 contains the sign 'digit'
      03F3 1452
      03F3 1453      CASE R0,TYPE=B,LIMIT=#10,<- ; Dispatch on sign digit
      03F3 1454      220$,- ; 10 => sign is '+'
      03F3 1455      210$,- ; 11 => sign is '-'
      03F3 1456      220$,- ; 12 => sign is '+'
      03F3 1457      210$,- ; 13 => sign is '-'
      03F3 1458      220$,- ; 14 => sign is '+'
      03F3 1459      220$,- ; 15 => sign is '+'
      03F3 1460      >
      54 01 D0 0403 1461 210$: MOVL #1,R4 ; Count a minus sign
      02 11 0406 1463      BRB 230$ ; Now check second input sign
      54 04 0408 1464
      040A 1465 220$: CLRL R4 ; No real minus signs so far
      52 52 08 AE 7D 040A 1467 230$: MOVQ B(SP),R2 ; Retrieve original R2/R3 pair
      52 04 01 EF 040E 1468      EXTZV #1,#4,R2,R2 ; Get byte count for second input string
      53 52 C0 0413 1469      ADDL R2,R3 ; Point R3 to byte containing sign
      52 63 F0 8F 8B 0416 1470      MARK_POINT MULP_DIVP_0
      0416 1471      BICB3 #*B11110000,(R3),R2 ; R2 contains the sign 'digit'
      041B 1472
      041B 1473      CASE R2,TYPE=B,LIMIT=#10,<- ; Dispatch on sign digit
      041B 1474      250$,- ; 10 => sign is '+'
      041B 1475      240$,- ; 11 => sign is '-'
      041B 1476      250$,- ; 12 => sign is '+'
      041B 1477      240$,- ; 13 => sign is '-'
      041B 1478      250$,- ; 14 => sign is '+'
      041B 1479      250$,- ; 15 => sign is '+'
      041B 1480      >
      042B 1481
      10 09 54 D6 042B 1482 240$: INCL R4 ; Remember that sign was minus
      58 02 E9 042D 1483 250$: BLBC R4,260$ ; Even parity indicates positive result
      58 08 88 0430 1484      BBS #PSLSV_Z,R11,270$ ; Step out of line for minus zero check
      56 D6 0434 1485      BISB #PSLSM_N,R11 ; Set N-bit in saved PSW
      0437 1486 255$: INCL R6 ; Change sign to minus
      0439 1487
      0439 1488
      67 04 00 56 F0 0439 1489 260$: MARK_POINT MULP_DIVP_0
      10 AE D4 043E 1490      INSV R6,#0,#4,(R7) ; Store sign in result string
      FBBC 31 0441 1491      CLRL 16(SP) ; Set saved R4 to zero
      0444 1492      BRW VAX$DECIMAL_EXIT ; Join common exit code
```

```

0444 1493 : If the result is negative zero, then it must be changed to positive zero
0444 1494 : unless overflow has occurred, in which case, the sign is left as negative
0444 1495 : but the N-bit is clear.
0444 1496
EF SB 01 E0 0444 1497 270$: BBS #PSL$V_V,R11,255$ ; Make sign negative if overflow
EF 11 0448 1498 BRB 260$ ; Sign will be positive

```

```
044A 1500 .SUBTITLE      EXTEND_STRING_MULTIPLY - Multiply a String by a Number
044A 1501 :+
044A 1502 Functional Description:
044A 1503
044A 1504 This routine multiplies an array of numbers (each array element LEQU
044A 1505 99) by a number (also LEQU 99). The resulting product array is added
044A 1506 to another array, each of whose elements is also LEQU 99.
044A 1507
044A 1508 Input Parameters:
044A 1509
044A 1510 R3 - Pointer to output array
044A 1511 R4 - Input array size
044A 1512 R5 - Input array address
044A 1513 R6 - Multiplier
044A 1514
044A 1515 Output Parameters:
044A 1516
044A 1517 None
044A 1518
044A 1519 Implicit Output:
044A 1520
044A 1521 The output array is altered.
044A 1522
044A 1523 An intermediate product array is produced by multiplying each input
044A 1524 array element by the multiplier. Each product array element is then
044A 1525 added to the corresponding output array element.
044A 1526
044A 1527 Side Effects:
044A 1528
044A 1529 R3, R4, and R5 are modified by this routine.
044A 1530
044A 1531 R6 is preserved.
044A 1532
044A 1533 R0, R1, and R2 are used as scratch registers. R0 and R1 contain the
044A 1534 quadword result of EMUL that is then passed into EDIV.
044A 1535
044A 1536 Assumptions:
044A 1537
044A 1538 This routine assumes that all array elements lie in the range from 0
044A 1539 to 99 inclusive. (This is true if all input strings contain only legal
044A 1540 decimal digits.) The arithmetic performed by this routine will
044A 1541 maintain this assumption. That is,
044A 1542
044A 1543      input array element      LEQU 99
044A 1544      times      multiplier      LEQU 99
044A 1545      -----
044A 1546      product
044A 1547      plus      carry      LEQU 99      LEQU 99+99
044A 1548      -----
044A 1549      modified product
044A 1550      plus      old output array element      LEQU 99      LEQU 99+100
044A 1551      -----
044A 1552      new output array element      LEQU 99+101 = 9999
044A 1553
044A 1554 A number LEQU 9999, when divided by 100, is guaranteed to produce both
044A 1555 a quotient and a remainder LEQU 99.
044A 1556 :-
```

```

      50  52  85  56  7A  044C  1561 10$: EMUL  R6,(R5)+,R2,R0      ; Form modified product (R0 LEQU 9900)
      50  52  50  00000064 8F  7B  0451 1562      ADDL2  (R3),R0      ; Add old output array element
83  52  50  00000064 8F  7B  0454 1563      EDIV   #100,R0,R2,(R3)+  ; Remainder to output array
      EC 54  F5  045D 1564      SOBGTR  R4,10$      ; Quotient becomes carry
      EC 54  F5  045D 1565      SOBGTR  R4,10$      ; Keep going?
      EC 54  F5  0460 1566
      EC 54  F5  0460 1567 : This remaining code looks more complicated than it actually is. In the
      EC 54  F5  0460 1568 : usual case, the routine exits immediately. In the event that a carry
      EC 54  F5  0460 1569 : occurs, one additional entry in the output array will be modified. Only in
      EC 54  F5  0460 1570 : the rare case of an output array consisting of a string of 99s will any
      EC 54  F5  0460 1571 : significant looping occur.
      EC 54  F5  0460 1572
      EC 54  F5  0460 1573      ADDL2  R2,(R3)      ; Add final carry
00000064 63  52  C0  0460 1573      CMPL   (R3),#100      ; Do we overflow into next digit pair?
      63  52  63  D1  0463 1574      BGEQU  30$      ; Branch if carry
      63  52  01  1E  046A 1575      RSB      30$      ; Otherwise, all done
      63  52  01  1E  046C 1576
      63  52  01  1E  046D 1577
83  00000064 8F  C2  046D 1577      30$:  SUBL   #100,(R3)+  ; Readjust entry and advance pointer
      63  52  63  D6  046D 1578      INCL   (R3)      ; Propagate carry
      63  52  63  D6  0474 1579      BRB    20$      ; ... and test this entry for overflow
      EB  11  0476 1580
```

```
0478 1582 .SUBTITLE VAX$DIVP - Divide Packed
0478 1583
0478 1584 :+ Functional Description:
0478 1585
0478 1586 The dividend string specified by the dividend length and dividend
0478 1587 address operands is divided by the divisor string specified by the
0478 1588 divisor length and divisor address operands. The quotient string
0478 1589 specified by the quotient length and quotient address operands is
0478 1590 replaced by the result.
0478 1591
0478 1592 Input Parameters:
0478 1593
0478 1594 R0 - divrlen.rw Number of digits in divisor string
0478 1595 R1 - divraddr.ab Address of divisor string
0478 1596 R2 - divdlen.rw Number of digits in dividend string
0478 1597 R3 - divdaddr.ab Address of dividend string
0478 1598 R4 - quolen.rw Number of digits in quotient string
0478 1599 R5 - quoaddr.ab Address of quotient string
0478 1600
0478 1601 Output Parameters:
0478 1602
0478 1603 R0 = 0
0478 1604 R1 = Address of the byte containing the most significant digit of
0478 1605 the divisor string
0478 1606 R2 = 0
0478 1607 R3 = Address of the byte containing the most significant digit of
0478 1608 the dividend string
0478 1609 R4 = 0
0478 1610 R5 = Address of the byte containing the most significant digit of
0478 1611 the string containing the quotient
0478 1612
0478 1613 Condition Codes:
0478 1614
0478 1615 N <- quotient string LSS 0
0478 1616 Z <- quotient string EQL 0
0478 1617 V <- decimal overflow
0478 1618 C <- 0
0478 1619
0478 1620 Register Usage:
0478 1621
0478 1622 This routine uses all of the general registers. The condition codes
0478 1623 are computed at the end of the instruction as the final result is
0478 1624 stored in the quotient string. R11 is used to record the condition
0478 1625 codes.
0478 1626
0478 1627 Algorithm:
0478 1628
0478 1629 This algorithm is the straightforward approach described in
0478 1630
0478 1631 The Art of Computer Programming
0478 1632 Second Edition
0478 1633
0478 1634 Volume 2 / Seminumerical Algorithms
0478 1635 Donald E. Knuth
0478 1636
0478 1637 1981
0478 1638 Addison-Wesley Publishing Company
```

```
0478 1639 : Reading, Massachusetts
0478 1640 :
0478 1641 : Notes:
0478 1642 :
0478 1643 : The choice of a longword array to store the auotient deserves a
0478 1644 : comment. In VAX$MULP, a longword array was used because its elements
0478 1645 : were used directly by MULP and DIVP instructions. The use of longwords
0478 1646 : eliminated the need to convert back and forth between longwords and
0478 1647 : bytes. In this routine, the QUOTIENT_DIGIT routine returns its result
0478 1648 : in a register, which result can easily be stored in whatever way is
0478 1649 : convenient. By using longwords instead of bytes, this routine can use
0478 1650 : the same end processing code as MULP, a sizeable savings in code.
0478 1651 :-
0478 1652 :
0478 1653 : .ENABLE LOCAL_BLOCK
0478 1654 :
0478 1655 : +
0478 1656 : This code path is entered if the divisor is zero.
0478 1657 :
0478 1658 : Input Parameter:
0478 1659 :
0478 1660 : (SP) - Return PC
0478 1661 :
0478 1662 : Output Parameters:
0478 1663 :
0478 1664 : 0(SP) - SRMSK_FLT_DIV_T (Arithmetic trap code)
0478 1665 : 4(SP) - Final state PSL
0478 1666 : 8(SP) - Return PC
0478 1667 :
0478 1668 : Implicit Output:
0478 1669 :
0478 1670 : Control passes through this code to VAX$REFLECT_TRAP.
0478 1671 :-
0478 1672 :
0478 1673 : DIVIDE_BY_ZERO:
OFFF 8F BA 0478 1674 : POPR #*M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>
0478 1675 : ; Restore registers and reset SP
0478 1676 : MOVPSL -(SP) ; Save final PSL on stack
0478 1677 : PUSHL #SRMSK_FLT_DIV_T ; Store arithmetic trap code
FB7D' 31 0480 1678 : BRW VAX$REFLECT_TRAP ; Report exception
0483 1679 :
0483 1680 : ; If the divisor contains more nonzero digits than the dividend, then the
0483 1681 : ; quotient will be identically zero. Set up the stack and the registers (R4,
0483 1682 : ; R5, and R9) so that the exit code will be entered to produce this result.
0483 1683 :
0483 1684 : 1$: CLRL -(SP) ; Fake a quotient digit
59 7E D4 0485 1685 : MOVL #1,R9 ; Count that digit
FECC 31 0488 1686 : BRW MULTIPLY_DIVIDE_EXIT ; Store the zero in the output string
0488 1687 :
0488 1688 : VAX$DIVP::
OFFF 8F BB 0488 1689 : PUSHR #*M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11> ; Save the lot
048F 1690 :
048F 1691 : ESTABLISH HANDLER - ; Store address of access
048F 1692 : ARITH_ACCVIO ; violation handler
0494 1693 :
0494 1694 : ROPRAND_CHECK R4 ; Insure that R4 is LEQU 31
049F 1695 :
```

```

      049F 1696      ROPRND CHECK R2      : Insure that R2 is LEQU 31
      04A7 1697      MARK_POINT DIVP BSBW 0
FB56' 30 04A7 1698      BSBW- DECIMAL$STRIP_ZEROS_R2_R3      : Strip high order zeros from R2/R3
      04AA 1699
      04AA 1700      ROPRND CHECK R0      : Insure that R0 is LEQU 31
      04B2 1701      MARK_POINT DIVP BSBW 0
FB4B' 30 04B2 1702      BSBW- DECIMAL$STRIP_ZEROS_R0_R1      : Strip high order zeros from R0/R1
      04B5 1703
      04B5 1704 : Insure that the divisor is not zero. Because leading zeros have already
      04B5 1705 : been eliminated, the divisor can only be zero if R0 is 0 (zero length
      04B5 1706 : strings are identically zero) or 1 (R1 contains a sign digit in the low
      04B5 1707 : order nibble and zero in the high order nibble). Note that an exception
      04B5 1708 : will not be generated if an even length string has an illegal nonzero digit
      04B5 1709 : stored in its most significant nibble (including an illegal form of a zero
      04B5 1710 : length string.
      04B5 1711
50 50 04 01 EF 04B5 1712      EXTZV #1,#4,R0,R0      : Convert divisor digit count to bytes
      06 12 04BA 1713      BNEQ 10$      : Skip zero divisor check unless zero
      04BC 1714      MARK_POINT DIVP 0
      61 F0 8F 93 04BC 1715      BITB #B11110000,(R1)      : Check for zero in ones digit
      B6 13 04C0 1716      BEQL DIVIDE_BY_ZERO      : Generate exception if zero
      04C2 1717
      04C2 1718 : This routine chooses to do its work with a fair amount of internal storage,
      04C2 1719 : all of it allocated on the stack. The quotient is stored as it is computed,
      04C2 1720 : in a 16-longword array. The dividend and divisor are stored as longword arrays,
      04C2 1721 : with each array element storing a digit pair from the original packed
      04C2 1722 : decimal string. The numerator digits are shifted by one digit (multiplied
      04C2 1723 : by ten) so that the quotient has its digits correctly placed, leaving room
      04C2 1724 : for a sign in the low order nibble of the least significant byte. A scratch
      04C2 1725 : array is also allocated on the stack to accommodate intermediate results
      04C2 1726 : of the QUOTIENT_DIGIT routine.
      04C2 1727
      58 50 D6 04C2 1728 10$: INCL R0      : Include least significant digit
      7D 04C4 1729      MOVQ R0,R8      : Let R8 and R9 describe the divisor
      04C7 1730
52 52 04 01 EF 04C7 1731      EXTZV #1,#4,R2,R2      : Convert dividend digit count to bytes
      52 D6 04CC 1732      INCL R2      : Include least significant digit
      7E 52 7D 04CE 1733      MOVQ R2,-(SP)      : Save dividend descriptor on stack
      04D1 1734
      56 52 50 C3 04D1 1735      SUBL3 R0,R2,R6      : Calculate main loop count
      AC 1F 04D5 1736      BLSSU 1$      : Quotient will be zero
      56 D6 04D7 1737      INCL R6      : One extra digit is always there
      04D9 1738
      04D9 1739 : Allocate R6 longwords of zero on the stack
      04D9 1740
      50 56 D0 04D9 1741      MOVL R6,R0      : Let R0 be the loop counter
      7E D4 04DC 1742 15$: CLRL -(SP)      : Set aside another quotient digit
      FB 50 F5 04DE 1743      SOBGTR R0,15$      : Keep going
      04E1 1744
      57 5E D0 04E1 1745      MOVL SP,R7      : Remember where this array starts
      04E4 1746
      04E4 1747 : The divisor will be stored on the stack as an array of
      04E4 1748 : longwords. Each array element contains a number between 0 and 99,
      04E4 1749 : representing a pair of digits in the original packed decimal string.
      04E4 1750 : Because the units digit is stored with the sign in packed decimal format,
      04E4 1751 : it is necessary to shift the number as we store it. This is accomplished by
      04E4 1752 : multiplying the number by ten.
```

				04E4	1753	:	
				04E4	1754	:	The divisor string is described by R8 (byte count) and R9 (address of most
				04E4	1755	:	significant digit pair).
				04E4	1756	:	
55	58	59	C1	04E4	1757		ADDL3 R9,R8,R5 ; Point R5 beyond sign digit
	54	58	D0	04E8	1758		MOVL R8,R4 ; R4 contains the loop count
				04EB	1759	:	
				04EB	1760	:	Put in an extra digit place for the divisor. This allows several common
				04EB	1761	:	subroutines to be used when operating on the divisor string.
				04EB	1762	:	
		7E	D4	04EB	1763		CLRL -(SP) ; Set aside a place holder
				04ED	1764	:	
				04ED	1765	:	An array of longwords is allocated on the stack. R3 starts out pointing
				04ED	1766	:	at the longword beyond the top of the stack. The first remainder, guaranteed
				04ED	1767	:	to be zero, is 'stored' here. The rest of the digit pairs are stored safely
				04ED	1768	:	below the top of the stack.
				04ED	1769	:	
	53	58	CE	04ED	1770		MNEGL R8,R3 ; Stack grows toward lower addresses
	5E	6E43	DE	04F0	1771		MOVAL (SP)[R3],SP ; Allocate the space
53	5E	04	C3	04F4	1772		SUBL3 #4,SP,R3 ; Point R3 at next lower longword
				04F8	1773	:	
				04F8	1774	:	MARK POINT DIVP_R6_R7
	51	75	9A	04F8	1775	20\$:	MOVZBL -(R5),R1 ; Get next digit pair
51	0000'	CF41	9A	04FB	1776		MOVZBL DECIMAL\$PACKED_TO_BINARY_TABLE[R1],-
				0501	1777		R1 ; Convert digits to binary
				0501	1778		EMUL #10,R1,R2,R0 ; Multiply by 10
B3	52	50		0506	1779		EDIV #100,R0,R2,(R3)+ ; Divide by 100
	50	52		050F	1780		SOBGR R4,20\$
	00000064	8F	7A	0501	1778		
	E6	54	F5	0506	1779		
				0512	1781	:	
				0512	1782	:	There are two cases where the final quotient (contents of R2) is zero.
				0512	1783	:	In these cases, the number of nonzero digit pairs in the divisor array is
				0512	1784	:	smaller by one than the number of bytes containing the original packed decimal
				0512	1785	:	string. One case is a divisor string with an even number of digits. The
				0512	1786	:	second case is a divisor string with an odd number of digits but the most
				0512	1787	:	significant digit is zero (essentially a variant of the first case). The
				0512	1788	:	simplest way to handle all of these cases is to decrement R8, the divisor
				0512	1789	:	counter, if R2 is zero. Note that previous checks for a zero divisor
				0512	1790	:	prevent R8 from going to zero.
				0512	1791	:	
63	52		D0	0512	1792		MOVL R2,(R3) ; Store final quotient
	0A		12	0515	1793		BNEQ 25\$; Leave well enough alone if nonzero
	56		D6	0517	1794		INCL R6 ; One more quotient digit
57	04		C2	0519	1795		SUBL #4,R7 ; Make room for it
	58		D7	051C	1796		DECL R8 ; Count one less divisor "digit"
	01		12	051E	1797		BNEQ 25\$
				0520	1798	:	
				0520	1799	:::	***** BEGIN TEMP *****
				0520	1800	:::	
				0520	1801	:::	THE FOLLOWING HALT INSTRUCTION SHOULD BE REPLACED WITH THE CORRECT
				0520	1802	:::	ABORT CODE.
				0520	1803	:::	
				0520	1804	:::	THE HALT IS SIMILAR TO THE
				0520	1805	:::	
				0520	1806	:::	MICROCODE CANNOT GET HERE
				0520	1807	:::	
				0520	1808	:::	ERRORS THAT OTHER IMPLEMENTATIONS USE.
				0520	1809	:::	

```
00 0520 1810      halt                      ; This will cause an OPCDEC exception
0521 1811 :::
0521 1812 ::: ***** END TEMP *****
59 5E D0 0521 1813 25$: MOVL SP,R9           ; R9 locates low order divisor digit
0524 1814
0524 1815 : The dividend is stored on the stack as an array of longwords. It does not
0524 1816 : have its digit pairs shifted so that this storage loop is simpler. An extra
0524 1817 : place is set aside in the event that it is necessary to normalize the
0524 1818 : dividend and divisor before division is attempted.
0524 1819
0524 1820
52 7E D4 0524 1821      CLRL -(SP)           ; Set aside space for U[0]
52 6746 DE 0526 1822      MOVAL (R7)[R6],R2      ; Retrieve dividend descriptor
52 62 7D 052A 1823      MOVQ (R2),R2           ; ... in two steps
052D 1824
052D 1825 MARK_POINT DIVP_R6_R7
7E 51 83 9A 052D 1826 30$: MOVZBL (R3)+,R1      ; Get next decimal digit pair
0000'CF41 9A 0530 1827      MOVZBL DECIMAL$PACKED_TO_BINARY, TABLE[R1],-
F4 52 F5 0536 1828      -(SP)                ; Convert digits to binary
0536 1829      SOBGTR R2,30$                 ; Loop through entire input string
0539 1830
0539 1831 : From this point until the common exit path for MULP and DIVP is entered,
0539 1832 : no access violations that need to be backed out can occur. We do not need
0539 1833 : to keep the address of ARITH_ACCVIO in R10 for this stretch of code. Note
0539 1834 : that R10 must be reloaded before the exit code executes because the
0539 1835 : destination string is written and may cause access violations.
0539 1836
5A 6746 D0 0539 1837      MOVL (R7)[R6],R10      ; Retrieve size of dividend array
5B 5E D0 053D 1838      MOVL SP,R11           ; R11 locates low order dividend digit
0540 1839
0540 1840 : Allocate a scratch array on the stack the same size as the divisor array
0540 1841 : (which is one larger than the number of digit pairs)
0540 1842
52 58 CE 0540 1843      MNEGL R8,R2           ; Need a negative index
5E FC AE42 DE 0543 1844      MOVAL -4(SP)[R2],SP ; Adjust stack pointer
0548 1845
0548 1846 :+
0548 1847 : At this point, the stack and relevant general registers contain the
0548 1848 : following information. In this description, N represents the number
0548 1849 : of digit pairs in the divisor and M represents the number of digit
0548 1850 : pairs in the dividend.
0548 1851
0548 1852 :
0548 1853 : scratch -----+ <-- SP
0548 1854 : | N+1 longwords |
0548 1855 : +-----+ <-- R11
0548 1856 : dividend -----+
0548 1857 : | M+1 longwords |
0548 1858 : +-----+ <-- R9
0548 1859 : divisor -----+
0548 1860 : | N+1 longwords |
0548 1861 : +-----+ <-- R7
0548 1862 : quotient -----+
0548 1863 : | M+1-N longwords |
0548 1864 : +-----+
0548 1865 : | R0..R11 |
0548 1866 : +-----+

R6 - Number of longwords in quotient array (M+1-N)
R7 - Address of beginning of quotient array
R8 - Number of digit pairs in divisor (called N)
```

```
0548 1867 : R9 - Address of low order digits in divisor
0548 1868 : R10 - Number of digit pairs in dividend (called M)
0548 1869 : R11 - Address of low order digits in dividend
0548 1870 :-
0548 1871 :
0548 1872 :
7E 6E DF 0548 1873 : PUSHAL (SP) : Store address of scratch array
7E 58 7D 054A 1873 : MOVQ R8,-(SP) : Remember divisor descriptor
7E 5A 7D 054D 1874 : MOVQ R10,-(SP) : Remember dividend descriptor
0550 1875 :
0550 1876 : The algorithm that guesses the quotient digit can be guaranteed to be off
0550 1877 : by no more than two if the high order digit of the divisor (called V[1]) is
0550 1878 : at least as large as 50 (our radix divided by 2). If the high order digit
0550 1879 : is too small, we "normalize" the numerator and denominator by multiplying
0550 1880 : them by the same number, namely 100/(V[1]+1).
0550 1881 :
50 FC A948 01 C1 0550 1882 : ADDL3 #1,-4(R9)[R8],R0 : Compute V[1] + 1
33 50 D1 0556 1883 : CMPL R0,#51 : Compare to 50 + 1
14 18 0559 1884 : BGEQ 40$ : Skip normalization if V[1] big enough
53 00000064 8F 50 C7 055B 1885 : DIVL3 R0,#100,R3 : Compute normalization factor
54 58 7D 0563 1886 : MOVQ R8,R4 : Get descriptor of divisor
00E0 30 0566 1887 : BSBW MULTIPLY_STRING : Normalize divisor
54 5A 7D 0569 1888 : MOVQ R10,R4 : Get descriptor of dividend
00DA 30 056C 1889 : BSBW MULTIPLY_STRING : Normalize dividend
056F 1890 :
056F 1891 : We have now reached the point where we can start calculating quotient digits.
056F 1892 : In the following loop, R5 and R6 are loop invariants. R5 contains the number
056F 1893 : of digit pairs in the divisor. R6 always points to the longword beyond the
056F 1894 : most significant digit in the dividend string. R7 and R8 must be loaded on
056F 1895 : each pass through because these two pointers are modified. Notice that the
056F 1896 : address of the divisor array is exactly what we want to store in R6.
056F 1897 :
5A 56 7D 056F 1898 40$: MOVQ R6,R10 : Let R10/R11 describe quotient and loop
5B 58 DD 0572 1899 : PUSHL R11 : Save quotient address for exit code
6B4A DE 0574 1900 : MOVAL (R11)[R10],R11 : Store quotient digits from high end
0578 1901 :
0578 1902 : This rather harmless looking loop is where the work is done
0578 1903 :
55 58 7D 0578 1904 : MOVQ R8,R5 : Initialize count and dividend address
59 5A D0 057B 1905 : MOVL R10,R9 : Remember the loop count in R9
057E 1906 :
57 10 AE 7D 057E 1907 50$: MOVQ 16(SP),R7 : Load divisor and scratch addresses
001F 30 0582 1908 : BSBW QUOTIENT_DIGIT : Get the next quotient digit
7B 53 D0 0585 1909 : MOVL R3,-(R11) : Store it
56 04 C2 0588 1910 : SUBL #4,R6 : "Advance" dividend pointer
F0 5A F5 058B 1911 : SOBGTR R10,50$ : ... and go back for more
058E 1912 :
058E 1913 : The quotient digits have been stored on the stack. Eliminate the rest of the
058E 1914 : stack storage and enter the completion code that this routine shares with
058E 1915 : VAX$MULP. Note that R9 is already set up with the longword count used by
058E 1916 : the exit code. Note also that R11 is pointing to the saved dividend descriptor
058E 1917 : that sits on top of the saved register array.
058E 1918 :
54 5E 6E D0 058E 1919 : MOVL (SP),SP : Reset stack pointer
18 AB49 DE 0591 1920 : MOVAL <<4*2> + - : Skip over saved dividend descriptor
54 64 7D 0596 1921 : <4*4> >(R11)[R9],R4 : and retrieve original R4 and R5
0596 1922 : MOVQ (R4),R4 : ... in two steps
0599 1923 :
```

```

0599 1924 : The following is a HACK.
0599 1925 :
0599 1926 : The method used to obtain quotient digits generally leaves garbage (nonzero)
0599 1927 : in what will become the sign digit. (In fact, this is the tenths digit of a
0599 1928 : decimal expansion of the remainder.) We need to make the least significant
0599 1929 : digit a multiple of ten.
0599 1930 :
50 6E 0A C7 0599 1931 DIVL3 #10,(SP),R0 ; Divide by ten, losing remainder
6E 50 0A C5 059D 1932 MULL3 #10,R0,(SP) ; Store only tens digit
FDB3 31 05A1 1933
05A1 1934 BRW MULTIPLY_DIVIDE_EXIT ; Join common exit code
05A4 1935
05A4 1936 .DISABLE LOCAL_BLOCK

```

05A4 1938 .SUBTITLE QUOTIENT_DIGIT - Get Next Digit in Quotient
05A4 1939 :
05A4 1940 : Functional Description:
05A4 1941 :
05A4 1942 : This routine divides an (N+1)-element array of longwords by an N-element
05A4 1943 : array, producing a single quotient digit in the range of 0 to 99
05A4 1944 : inclusive. The dividend array is modified by subtracting the product
05A4 1945 : of the divisor array and the quotient digit.
05A4 1946 :
05A4 1947 : The "numbers" that this array operates on multiple precision numbers
05A4 1948 : in radix 100. Each digit (a number between 0 and 99) is stored in a
05A4 1949 : longword array element with more significant digits stored at higher
05A4 1950 : addresses. The dividend string and the scratch string (also called the
05A4 1951 : product string) contain one more element than the divisor string.
05A4 1952 :
05A4 1953 : Input Parameters:
05A4 1954 :
05A4 1955 : R5 - Number of "digits" (array elements) in divisor array (preserved)
05A4 1956 : R6 - Address of longword immediately following most significant
05A4 1957 : digit of dividend string (preserved)
05A4 1958 : R7 - Address of least significant digit in divisor string (modified)
05A4 1959 : R8 - Address of least significant digit in product string (modified)
05A4 1960 :
05A4 1961 : Output Parameters:
05A4 1962 :
05A4 1963 : R3 - The quotient that results from dividing the dividend string
05A4 1964 : by the divisor string.
05A4 1965 :
05A4 1966 : The final states of the three pointer registers are listed here
05A4 1967 : for completeness.
05A4 1968 :
05A4 1969 : R6 - Address of longword immediately following most significant
05A4 1970 : digit of dividend string
05A4 1971 :
05A4 1972 : R7 - Address of longword immediately following most significant digit
05A4 1973 : of divisor string. This longword must always contain zero.
05A4 1974 :
05A4 1975 : R8 - Address of longword immediately following most significant
05A4 1976 : digit of product string
05A4 1977 :
05A4 1978 : Implicit Output:
05A4 1979 :
05A4 1980 : The contents of the dividend array are modified to reflect the
05A4 1981 : subtraction of the product string. The result of this subtraction
05A4 1982 : could be stored elsewhere. It is a convenience to store it in the
05A4 1983 : dividend array on top of those array elements that are no longer
05A4 1984 : needed.
05A4 1985 :
05A4 1986 : The contents of the divisor array are preserved.
05A4 1987 :
05A4 1988 : Side Effects:
05A4 1989 :
05A4 1990 : R7 and R8 are modified by this routine. (See implicit output list.)
05A4 1991 :
05A4 1992 : R5 and R6 are preserved.
05A4 1993 :
05A4 1994 : R0, R1, R2, and R4 are used as scratch registers. R0 and R1 contain the

```
05A4 1995 : quadword result of EMUL that is then passed into EDIV. R2 is the
05A4 1996 : carry from one step to the next. R4 is the loop counter.
05A4 1997 :-
05A4 1998
05A4 1999 QUOTIENT DIGIT:
FB A6 FC A6 00000064 8F 7A 05A4 2000 EMUL #100,-4(R6),-8(R6),R0 ; R0 <- 100 * U[j] + U[j+1]
50 50 05AE
50 FC A745 C6 05AF 2001 DIVL2 -4(R7)[R5],R0 ; R0 <- R0 / V[1]
53 50 D0 05B4 2002 MOVL R0,R3 ; Store quotient 'digit' in R3
65 13 05B7 2003 BEQL 45$ ; Nothing to do if quotient is zero
00000064 8F 53 D1 05B9 2004 CMPL R3,#100 ; Is quotient LEQU 99?
07 1F 05C0 2005 BLSSU 5$ ; Branch if quotient OK
53 00000063 8F D0 05C2 2006 MOVL #99,R3 ; Otherwise start with 99
05C9 2007
05C9 2008 : We will now multiply the divisor array by the quotient digit, storing the
05C9 2009 : product in the scratch array.
05C9 2010
05C9 2011 5$: CLRL R2 ; Start out with a carry of zero
54 55 D0 05CB 2012 MOVL R5,R4 ; R4 will be the loop counter
05CE 2013
05CE 2014 10$: EMUL R3,(R7)+,R2,R0 ; Multiply next divisor digit
88 52 50 52 87 53 7A 05D3 2015 EDIV #100,R0,R2,(R8)+ ; Remainder to input array
05DC 2016 ; Quotient becomes carry
EF 54 F5 05DC 2017 SOBGTR R4,10$ ; More divisor digits?
05DF 2018
88 52 D0 05DF 2019 MOVL R2,(R8)+ ; Store final carry
05E2 2020
05E2 2021 : If the product array is larger than the dividend array, then the quotient is
05E2 2022 : too large. To avoid a second trip through the rather costly EMUL/EDIV loop,
05E2 2023 : and also to avoid array subtraction that produces a negative result, we will
05E2 2024 : first compare the product and dividend arrays. If the product is smaller, we
05E2 2025 : can safely subtract. If the product is larger, we decrease the quotient by
05E2 2026 : one and subtract the divisor array from the product array.
05E2 2027
50 56 D0 05E2 2028 15$: MOVL R6,R0 ; Point R0 and R1 to high address ends
51 58 D0 05E5 2029 MOVL R8,R1 ; ... of dividend and scratch strings
54 55 D0 05E8 2030 MOVL R5,R4 ; Initialize the loop counter
05EB 2031
05EB 2032 : The comparison is done from most to least significant digits
05EB 2033
70 71 D1 05EB 2034 20$: CMPL -(R1),-(R0) ; Compare next pair of digits
OE 1F 05EE 2035 BLSSU 30$ ; Leave loop if product is smaller
2D 1A 05F0 2036 BGTRU 50$ ; Also leave if product is larger
F6 54 F4 05F2 2037 SOBGEQ R4,20$ ; More to test?
05F5 2038
05F5 2039 : If we drop through the loop, then the dividend and product are equal. We
05F5 2040 : simply store zeros in the dividend array (the equivalent of subtraction
05F5 2041 : of equal arrays) and return. Note that R0 is already pointing to the
05F5 2042 : least significant dividend array element.
05F5 2043
54 55 D0 05F5 2044 MOVL R5,R4 ; Initialize still another loop counter
05F8 2045
80 D4 05F8 2046 25$: CLRL (R0)+ ; Store another zero
FB 54 F4 05FA 2047 SOBGEQ R4,25$ ; Keep going?
05FD 2048
05FD 2049 ; Return to caller
05FE 2050
```

```
05FE 2051 : If we drop through the loop, then the quotient that is stored in R3 is good.
05FE 2052 : We need to subtract the product array from the dividend array. Note that R0
05FE 2053 : and R1 need to be adjusted to point to the least significant array elements
05FE 2054 : before the subtraction can begin.
05FE 2055
05FE 2056 30$: MNEGL R4,R4 : We need a negative index
0601 2057 MOVAL (R0)[R4],R0 : Adjust dividend pointer
0605 2058 MOVAL (R1)[R4],R1 : ... and product pointer
0609 2059 MOVL R5,R4 : R4 will count still another loop
060C 2060
060C 2061 35$: SUBL2 (R1)+,(R0)+ : Subtract next digits
060F 2062 BGEQ 40$ : Skip to end of loop if no borrow
0611 2063 ADDL2 #100,-4(R0) : Add borrow back to this digit
0619 2064 DECL (R0) : ... and borrow from next highest digit
061B 2065 40$: SOBGEQ R4,35$ : Keep going?
061E 2066
061E 2067 : This is the exit path. R3 contains the quotient digit. The pointers to the
061E 2068 : various input and scratch arrays are in an indeterminate state.
061E 2069
05 061E 2070 45$: RSB : Return to caller
061F 2071
061F 2072 : The first guess at the quotient digit is too large. The brute force
061F 2073 : approach is to decrement the quotient by one and execute the EMUL/EDIV loop
061F 2074 : again. Note, however, that we can evaluate the modified product by
061F 2075 : subtracting the divisor from the initial product. Note also that, because
061F 2076 : the leading digit in the divisor is "large enough", we can only end up in
061F 2077 : this code path twice. (That is, the initial guess at the quotient will
061F 2078 : never be off by more than two.)
061F 2079
053 061F 2080 50$: DECL R3 : Try quotient smaller by one
FB 13 0621 2081 BEQL 45$ : All done if zero
0623 2082
0623 2083 : Point R1 and R2 at the least significant digits of the scratch and product
0623 2084 : strings respectively.
0623 2085
05 0623 2086 MNEGL R5,R0 : Need a negative index
S1 50 55 CE 0626 2087 MOVAL -4(R8)[R0],R1 : Scratch array contains N+1 elements
FC A840 DE 062B 2088 MOVAL (R7)[R0],R2 : Product array contains N elements
52 6740 DE 062F 2089 MOVL R5,R4 : R4 will count still another loop
54 55 DO 0632 2090
0632 2091 60$: SUBL2 (R2)+,(R1)+ : Subtract next digits
0632 2092 BGEQ 70$ : Skip to end of loop if no borrow
0635 2093 ADDL2 #100,-4(R1) : Add borrow back to this digit
0637 2094 DECL (R1) : ... and borrow from next highest digit
063F 2095 70$: SOBGEQ R4,60$ : Keep going?
0641 2096
0644 2097
05 0644 2097 ADDL2 #4,R1 : Point R1 at most significant digit
99 11 0647 2098 BRB 15$ : Make another comparison
```

```
0649 2100 .SUBTITLE MULTIPLY_STRING - Multiply a String by a Number
0649 2101
0649 2102 Functional Description:
0649 2103
0649 2104 This routine multiplies an array of numbers (each array element LEQU
0649 2105 99) by a number (also LEQU 99). Each array element in the input array
0649 2106 is replaced with the modified product, with the carry propagated to
0649 2107 the next array element.
0649 2108
0649 2109 Input Parameters:
0649 2110
0649 2111 R3 - Multiplier
0649 2112 R4 - Input array size
0649 2113 R5 - Input array address
0649 2114
0649 2115 Output Parameters:
0649 2116
0649 2117 None
0649 2118
0649 2119 Implicit Output:
0649 2120
0649 2121 The input array elements are altered.
0649 2122
0649 2123 Side Effects:
0649 2124
0649 2125 R4 and R5 are modified by this routine.
0649 2126
0649 2127 R3 is preserved.
0649 2128
0649 2129 R0, R1, and R2 are used as scratch registers. R0 and R1 contain the
0649 2130 quadword result of EMUL that is then passed into EDIV. R2 is the
0649 2131 carry from one step to the next.
0649 2132
0649 2133 Assumptions:
0649 2134
0649 2135 This routine assumes that all array elements lie in the range from 0
0649 2136 to 99 inclusive. (This is true if all input strings contain only legal
0649 2137 decimal digits.) The arithmetic performed by this routine will
0649 2138 maintain this assumption. The details of this argument can be found in
0649 2139 the routine header for EXTENDED MULTIPLY_STRING. This routine performs
0649 2140 less work so that those arguments also apply here.
0649 2141
0649 2142
0649 2143 MULTIPLY_STRING:
0649 2144 CLRL R2 ; Initial carry is zero
0649 2145
0649 2146 10$: EMUL R3,(R5),R2,R0 ; Form modified product (R0 LEQU 9900)
0649 2147 EDIV #100,R0,R2,(R5)+ ; Remainder to input array
0649 2148 ; Quotient becomes carry
0649 2149 SOBGTR R4,10$ ; Keep going?
0649 2150
0649 2151 MOVL R2,(R5) ; Store final carry
0649 2152 RSB
```

```
52 D4
85 52 50 52 65 53 7A
50 00000064 8F 7B
EF 54 F5
65 52 D0
05 065F
```

```
0660 2154 .SUBTITLE DECIMAL_ROPRAND
0660 2155
0660 2156 :- Functional Description:
0660 2157
0660 2158 This routine receives control when a digit count larger than 31
0660 2159 is detected. The exception is architecturally defined as an
0660 2160 abort so there is no need to store intermediate state. All of the
0660 2161 routines in this module save all registers R0 through R11 before
0660 2162 performing the digit check. These registers must be restored
0660 2163 before control is passed to VAX$ROPRAND.
0660 2164
0660 2165 Input Parameters:
0660 2166
0660 2167 00(SP) - Saved R0
0660 2168 .
0660 2169
0660 2170 44(SP) - Saved R11
0660 2171 48(SP) - Return PC from VAX$xxxxxx routine
0660 2172
0660 2173 Output Parameters:
0660 2174
0660 2175 00(SP) - Offset in packed register array to delta PC byte
0660 2176 04(SP) - Return PC from VAX$xxxxxx routine
0660 2177
0660 2178 Implicit Output:
0660 2179
0660 2180 This routine passes control to VAX$ROPRAND where further
0660 2181 exception processing takes place.
0660 2182 :-
0660 2183
0660 2184 ASSUME ADDP6_B_DELTA_PC EQ ADDP4_B_DELTA_PC
0660 2185 ASSUME SUBP4_B_DELTA_PC EQ ADDP4_B_DELTA_PC
0660 2186 ASSUME SUBP6_B_DELTA_PC EQ ADDP4_B_DELTA_PC
0660 2187 ASSUME MULP_B_DELTA_PC EQ ADDP4_B_DELTA_PC
0660 2188 ASSUME DIVP_B_DELTA_PC EQ ADDP4_B_DELTA_PC
0660 2189
0660 2190 DECIMAL_ROPRAND:
OFFF 8F BA 0660 2191 POPR #*M<R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>
03 DD 0664 2192 PUSHL #ADDP4_B_DELTA_PC ; Store offset to delta PC byte
F997 31 0666 2193 BRW VAX$ROPRAND ; Pass control along
```

```
0669 2195 .SUBTITLE ARITH_ACCVIO - Reflect an Access Violation
0669 2196
0669 2197 :+ Functional Description:
0669 2198
0669 2199 This routine receives control when an access violation occurs while
0669 2200 executing within the emulator routines for ADDP4, ADDP6, SUBP4, SUBP6,
0669 2201 MULP, or DIVP.
0669 2202
0669 2203 The routine header for ASHP_ACCVIO in module VAX$ASHP contains a
0669 2204 detailed description of access violation handling for the decimal
0669 2205 string instructions.
0669 2206
0669 2207 : Input Parameters:
0669 2208
0669 2209 See routine ASHP_ACCVIO in module VAX$ASHP
0669 2210
0669 2211 : Output Parameters:
0669 2212
0669 2213 See routine ASHP_ACCVIO in module VAX$ASHP
0669 2214 :-
0669 2215
0669 2216 ARITH_ACCVIO:
0669 2217 CLRL R2 ; Initialize the counter
0669 2218 PUSHAB MODULE_BASE ; Store base address of this module
0669 2219 PUSHAB MODULE_END ; Store module end address
0669 2220 BSBW DECIMAL$BOUNDS_CHECK ; Check if PC is inside the module
0669 2221 ADDL #4,SP ; Discard end address
0669 2222 SUBL2 (SP)+,R1 ; Get PC relative to this base
0669 2223
0669 2224 10$: CMPW R1,PC_TABLE_BASE[R2] ; Is this the right PC?
0669 2225 BEQL 30$ ; Exit loop if true
0669 2226 AOBLS #TABLE_SIZE,R2,10$ ; Do the entire table
0669 2227
0669 2228 ; If we drop through the dispatching based on PC, then the exception is not
0669 2229 ; one that we want to back up. We simply reflect the exception to the user.
0669 2230
0669 2231 20$: POPR #*M<R0,R1,R2,R3> ; Restore saved registers
0669 2232 RSB ; Return to exception dispatcher
0669 2233
0669 2234 ; The exception PC matched one of the entries in our PC table. R2 contains
0669 2235 ; the index into both the PC table and the handler table. R1 has served
0669 2236 ; its purpose and can be used as a scratch register.
0669 2237
0669 2238 30$: MOVZWL HANDLER_TABLE_BASE[R2],R1 ; Get the offset to the handler
0669 2239 JMP MODULE_BASE[R1] ; Pass control to the handler
0669 2240
0669 2241 ; In all of the instruction-specific routines, the state of the stack
0669 2242 ; will be shown as it was when the exception occurred. All offsets will
0669 2243 ; be pictured relative to R0.
```

52 D4 0669 2217
F991 CF 9F 0668 2218
06D3'CF 9F 066F 2219
F98A' 30 0673 2220
SE 04 C0 0676 2221
51 8E C2 0679 2222
0000'CF42 51 B1 067C 2223
07 13 0682 2224
F4 52 28 F2 0684 2225
0688 2226
0688 2227
0688 2228
0688 2229
0688 2230
OF BA 0688 2231
05 068A 2232
0688 2233
0688 2234
0688 2235
0688 2236
0688 2237
51 0000'CF42 3C 0688 2238
F96A CF41 17 0691 2239
0696 2240
0696 2241
0696 2242
0696 2243

```
0696 2245 .SUBTITLE Access Violation Handling for ADDPx and SUBPx
0696 2246 :+
0696 2247 : Functional Description:
0696 2248 :
0696 2249 : The only difference among the various entry points is the number of
0696 2250 : longwords on the stack. R0 is advanced beyond these longwords to point
0696 2251 : to the list of saved registers. These registers are then restored,
0696 2252 : effectively backing the routine up to its initial state.
0696 2253 :
0696 2254 : Input Parameters:
0696 2255 :
0696 2256 : R0 - Address of top of stack when access violation occurred
0696 2257 :
0696 2258 : See specific entry points for details
0696 2259 :
0696 2260 : Output Parameters:
0696 2261 :
0696 2262 : See input parameter list for VAX$DECIMAL_ACCVIO in module VAX$ASHP
0696 2263 :-
0696 2264 :
0696 2265 :+
0696 2266 : ADD_SUB_BSBW_24
0696 2267 :
0696 2268 : An access violation occurred in one of the subroutines ADD_PACKED_BYTE,
0696 2269 : SUB_PACKED_BYTE, or STORE_RESULT. In addition to the six longwords of work
0696 2270 : space, this routine has an additional longword, the return PC, on the
0696 2271 : stack.
0696 2272 :
0696 2273 : 00(R0) - Return PC in mainline VAX$xxxxxx routine
0696 2274 : 04(R0) - Address of sign byte of destination string
0696 2275 : 08(R0) - First longword of scratch space
0696 2276 : etc.
0696 2277 :-
0696 2278 :
0696 2279 : ADD_SUB_BSBW_24:
50 04 C0 0696 2280 ADDL #4,R0 ; Skip over return PC and drop into ...
0699 2281 :+
0699 2282 : ADD_SUB_24
0699 2283 :
0699 2284 : There are five longwords of workspace and a saved string address on the stack
0699 2285 : for this entry point.
0699 2286 :
0699 2287 : 00(R0) - Address of sign byte of destination string
0699 2288 : 04(R0) - First longword of scratch space
0699 2289 :
0699 2290 :
0699 2291 :
0699 2292 : 20(R0) - Fifth longword of scratch space
0699 2293 : 24(SP) - Saved R0
0699 2294 : 28(SP) - Saved R1
0699 2295 : etc.
0699 2296 :-
0699 2297 :
0699 2298 : ADD_SUB_24:
50 18 C0 0699 2299 ADDL #24,R0 ; Discard scratch space on stack
F961' 31 069C 2300 BRW VAX$DECIMAL_ACCVIO ; Join common code to restore registers
069F 2301
```

```

069F 2302 :+
069F 2303 : ADD_SUB_BSBW_0
069F 2304 :
069F 2305 : An access violation occurred in one of the subroutine STRIP ZEROS. This
069F 2306 : entry point has an additional longword, the return PC, on the stack on top
069F 2307 : of the saved register array.
069F 2308 :
069F 2309 : 00(R0) - Return PC in mainline VAX$xxxxxx routine
069F 2310 : 04(R0) - Saved R0
069F 2311 : 08(R0) - Saved R1
069F 2312 : etc.
069F 2313 :-
069F 2314 :
069F 2315 ADD_SUB_BSBW_0:
50 04 C0 069F 2316 ADDL #4,R0 ; Skip over return PC and ...
F95B' 31 06A2 2317 BRW VAX$DECIMAL_ACCVIO ; Join common code to restore registers

```

```
06A5 2319 .SUBTITLE Access Violation Handling for MULP and DIVP
06A5 2320
06A5 2321 :+ Functional Description:
06A5 2322 :
06A5 2323 : The only difference among the various entry points is the number of
06A5 2324 : longwords on the stack. R0 is advanced beyond these longwords to point
06A5 2325 : to the list of saved registers. These registers are then restored,
06A5 2326 : effectively backing the routine up to its initial state.
06A5 2327 :
06A5 2328 : Input Parameters:
06A5 2329 :
06A5 2330 : R0 - Address of top of stack when access violation occurred
06A5 2331 :
06A5 2332 : See specific entry points for details
06A5 2333 :
06A5 2334 : Output Parameters:
06A5 2335 :
06A5 2336 : See input parameter list for VAX$DECIMAL_ACCVIO in module VAX$ASHP
06A5 2337 :-
06A5 2338 :
06A5 2339 :+
06A5 2340 : MULP_R8
06A5 2341 :
06A5 2342 : An access violation occurred while MULP was accessing one of its two source
06A5 2343 : strings. In this particular case, MULP was storing the longer of the two
06A5 2344 : input strings in a longword array on the top of the stack. There is an
06A5 2345 : array of R8 longwords on top of an array of 32 longwords on top of the
06A5 2346 : saved register array.
06A5 2347 :
06A5 2348 : R8 - Number of longwords on top of the 32-longword array
06A5 2349 :-
06A5 2350 :
06A5 2351 .ENABLE LOCAL_BLOCK
06A5 2352
06A5 2353 MULP_R8:
06A5 2354 MOVAL (R0)[R8],R0 ; Discard input array storage
06A9 2355 BRB 10$ ; Might as well share a little code
06AB 2356
06AB 2357 :+
06AB 2358 : MULP_AT_SP
06AB 2359 :
06AB 2360 : An access violation occurred while MULP was accessing one of its two source
06AB 2361 : strings. In this case, the access violation occurred in the middle of the
06AB 2362 : grand multiply loop as a digit pair was being retrieved from the shorter of
06AB 2363 : the two input strings. The address of the start of the 32-longword array
06AB 2364 : was itself stored on top of the stack for convenience.
06AB 2365 :
06AB 2366 : 00(R0) - Saved byte count of longer input string
06AB 2367 : 04(R0) - Saved address of longer input string
06AB 2368 : 08(R0) - Address of 32-longword array farther down the stack
06AB 2369 :-
06AB 2370 :
06AB 2371 MULP_AT_SP:
06AB 2372 -MOVL 8(R0),R0 ; Locate start of 32-longword array
06AF 2373 10$: MOVAB <<4*32> + <4*2>>(R0),R0 ; Throw that away, too
06B4 2374 BRW VAX$DECIMAL_ACCVIO ; Join common code to restore registers
06B7 2375
```

50 6048 DE 04 11

50 08 A0 DO 06AB 2372
50 0088 C0 9E 06AF 2373
F949 31 06B4 2374
06B7 2375

```
06B7 2376      .DISABLE      LOCAL_BLOCK
06B7 2377
06B7 2378      :+
06B7 2379      : Mulp_Divp_R9
06B7 2380      :
06B7 2381      : An access violation occurred while the final result was being stored in the
06B7 2382      : result string. In this common exit code path, R9 counts the number of
06B7 2383      : longwords on the stack. In all cases where an access violation can occur, a
06B7 2384      : longword has been removed from the stack but R9 has not yet been
06B7 2385      : decremented to reflect this. The conceptual instruction sequence that
06B7 2386      : resets the stack pointer (really R0) to point to the start of the saved
06B7 2387      : register array is
06B7 2388
06B7 2389      DECL    R9
06B7 2390      MOVAL   (R0)[R9]
06B7 2391
06B7 2392      A single instruction accomplishes this.
06B7 2393
06B7 2394      R9 - One more than the number of longwords on the stack on top
06B7 2395      of the saved register array.
06B7 2396
06B7 2397      00(R0) - First longword of scratch storage remaining on the stack
06B7 2398      .
06B7 2399      .
06B7 2400      zz-4(R0) - Last longword of scratch storage
06B7 2401      zz+0(R0) - Saved count of dividend or multiplier string
06B7 2402      zz+4(R0) - Saved address of dividend or multiplier string
06B7 2403      zz+8(R0) - Saved R0
06B7 2404      zz+12(R0) - Saved R1
06B7 2405      etc.
06B7 2406
06B7 2407      where zz = 4 * (R9 - 1)
06B7 2408      :-
06B7 2409
06B7 2410      Mulp_Divp_R9:
50 04 A049 DE 06B7 2411      MOVAL   4(R0)[R9],R0      : Discard scratch storage on stack
   F941' 31 06BC 2412      BRW     VAX$DECIMAL_ACCVIO    : Join common code to restore registers
06BF 2413
06BF 2414      :+
06BF 2415      : Mulp_Divp_8
06BF 2416      :
06BF 2417      : An access violation occurred in the common exit path after the scratch array
06BF 2418      : had been removed from the stack but before the saved descriptor for the
06BF 2419      : multiplier string was discarded.
06BF 2420
06BF 2421      0(R0) - Saved count of dividend or multiplier string
06BF 2422      4(R0) - Saved address of dividend or multiplier string
06BF 2423      8(R0) - Saved R0
06BF 2424      12(R0) - Saved R1
06BF 2425      etc.
06BF 2426      :-
06BF 2427
06BF 2428      Mulp_Divp_8:
50 08 C0 06BF 2429      ADDL    #8,R0      : Discard multiplier string descriptor
   F93B' 31 06C2 2430      BRW     VAX$DECIMAL_ACCVIO    : Join common code to restore registers
06C5 2431
06C5 2432      :+
```

```
06C5 2433 : MULP_BSBW_0
06C5 2434 : DIVP_BSBW_0
06C5 2435 :
06C5 2436 : An access violation occurred in one of the subroutine STRIP ZEROS. This
06C5 2437 : entry point has an additional longword, the return PC, on the stack on top
06C5 2438 : of the saved register array.
06C5 2439 :
06C5 2440 : 00(R0) - Return PC in mainline VAX$MULP or VAX$DIVP routine
06C5 2441 : 04(R0) - Saved R0
06C5 2442 : 08(R0) - Saved R1
06C5 2443 : etc.
06C5 2444 : -
06C5 2445 :
06C5 2446 : MULP_BSBW_0:
06C5 2447 : DIVP_BSBW_0:
50 04 C0 06C5 2448 : ADDL #4,R0 ; Skip over return PC and drop into ...
06C8 2449 :
06C8 2450 : +
06C8 2451 : DIVP_0
06C8 2452 : MULP_DIVP_0
06C8 2453 :
06C8 2454 : There was nothing allocated on the stack other than the saved register
06C8 2455 : array when the access violation occurred. We merely pass control to common
06C8 2456 : code to restore the registers.
06C8 2457 :
06C8 2458 : 00(R0) - Saved R0
06C8 2459 : 04(R0) - Saved R1
06C8 2460 : etc.
06C8 2461 : -
06C8 2462 :
06C8 2463 : DIVP_0:
06C8 2464 : MULP_DIVP_0:
F935' 31 06C8 2465 : BRW VAX$DECIMAL_ACCVIO ; Join common code to restore registers
06CB 2466 :
06CB 2467 : +
06CB 2468 : DIVP_R6_R7
06CB 2469 :
06CB 2470 : An access violation occurred while one of the two input strings was being
06CB 2471 : converted to an array of longwords on the stack. The state of the stack
06CB 2472 : is rather complicated but R6 and R7 contain enough information to allow
06CB 2473 : the rest of the stack contents to be ignored.
06CB 2474 :
06CB 2475 : R6 - Count of longwords in quotient array on stack
06CB 2476 : R7 - Address of quotient array on stack
06CB 2477 :
06CB 2478 : 00(R0) - First longword of quotient array
06CB 2479 : .
06CB 2480 : .
06CB 2481 : zz-4(R0) - Last longword of scratch storage
06CB 2482 : zz+0(R0) - Digit count of dividend string
06CB 2483 : zz+4(R0) - Address of dividend string
06CB 2484 : zz+8(R0) - Saved R0
06CB 2485 : zz+12(R0) - Saved R1
06CB 2486 : etc.
06CB 2487 :
06CB 2488 : where zz = 4 * R6
06CB 2489 : -
```

50	08	A746	DE	06CB	2490			
		F92D	31	06CB	2491	DIVP_R6_R7:		
				06CB	2492	MOVAL	8(R7)[R6],R0	: Discard everything on stack
				06D0	2493	BRW	VAX\$DECIMAL_ACCVIO	: Join common code to restore registers
				06D3	2494			
				06D3	2495	END_MARK_POINT		
				06D3	2496			
				06D3	2497	.END		

VAX\$DECIMAL_ARITHMETIC
Symbol table- VAX-11 Packed Decimal Arithmetic Instr 16-SEP-1984 01:33:44 VAX/VMS Macro V04-00
5-SEP-1984 00:44:34 [EMULAT.SRC]VAXARITH.MAR;1Page 55
(23)

```

...PC... = 0000052D
...ROPRAND... = 00000499 R 02
ADDP4_B_DELTA_PC = 00000003
ADDP6_B_DELTA_PC = 00000003
ADD_PACKED = 000000DA R 02
ADD_PACKED_BYTE_R6_R7 = 00000165 R 02
ADD_PACKED_BYTE_STRING = 0000015F R 02
ADD_SUBTRACT_EXIT = 00000132 R 02
ADD_SUB_24 = 00000699 R 02
ADD_SUB_BSBW_0 = 0000069F R 02
ADD_SUB_BSBW_24 = 00000696 R 02
ADD_SUB_V_ZERO_R4 = 0000001F
ARITH_ACCVIO = 00000669 R 02
DECIMAL$BINARY_TO_PACKED_TABLE = ***** X 00
DECIMAL$BOUNDS_CHECK = ***** X 00
DECIMAL$PACKED_TO_BINARY_TABLE = ***** X 00
DECIMAL$STRIP_ZEROS_R0_RT = ***** X 00
DECIMAL$STRIP_ZEROS_R2_R3 = ***** X 00
DECIMAL_ROPRAND = 00000660 R 02
DIVIDE_BY_ZERO = 00000478 R 02
DIVP_0 = 000006C8 R 02
DIVP_BSBW_0 = 000006C5 R 02
DIVP_B_DELTA_PC = 00000003
DIVP_R6_R7 = 000006CB R 02
EXTEND_STRING_MULTIPLY = 0000044A R 02
HANDLER_TABLE_BASE = 00000000 R 04
MODULE_BASE = 00000000 R 02
MODULE_END = 000006D3 R 02
MULP_AT_SP = 000006AB R 02
MULP_BSBW_0 = 000006C5 R 02
MULP_B_DELTA_PC = 00000003
MULP_DIVP_0 = 000006C8 R 02
MULP_DIVP_8 = 000006BF R 02
MULP_DIVP_R9 = 000006B7 R 02
MULP_R8 = 000006A5 R 02
MULTIPLY_DIVIDE_EXIT = 00000357 R 02
MULTIPLY_STRING = 00000649 R 02
PC_TABLE_BASE = 00000000 R 03
PSL$M_N = 00000008
PSL$M_V = 00000002
PSL$M_Z = 00000004
PSL$V_CM = 0000001F
PSL$V_V = 00000001
PSL$V_Z = 00000002
QUOTIENT_DIGIT = 000005A4 R 02
SRMSK_FLT_DIV_T = 00000004
STORE_RESULT = 00000249 R 02
SUBP4_B_DELTA_PC = 00000003
SUBP6_B_DELTA_PC = 00000003
SUBTRACT_PACKED = 0000018D R 02
SUB_PACKED_BYTE_R6_R7 = 00000223 R 02
SUB_PACKED_BYTE_STRING = 0000021D R 02
TABLE_SIZE = 00000028
VAX$ADDP4 = 0000002B RG 02
VAX$ADDP6 = 00000009 RG 02
VAX$ADD_PACKED_BYTE_R6_R7 = 00000165 RG 02
VAX$DECIMAL_ACCVIO = ***** X 00

```

```

VAX$DECIMAL_EXIT ***** X 00
VAX$DIVP 0000048B RG 02
VAX$MULP 00000287 RG 02
VAX$REFLECT_TRAP ***** X 00
VAX$ROPRAND ***** X 00
VAX$SUBP4 00000022 RG 02
VAX$SUBP6 00000000 RG 02

```

+-----+
! Psect synopsis !
+-----+

PSECT name	Allocation	PSECT No.	Attributes
. ABS .	00000000 (0.)	00 (0.)	NOPIC USR CON ABS LCL NOSHR NOEXE NORD NOWRT NOVEC BYTE
\$ABSS	00000000 (0.)	01 (1.)	NOPIC USR CON ABS LCL NOSHR EXE RD WRT NOVEC BYTE
VAX\$CODE	00000603 (1747.)	02 (2.)	PIC USR CON REL LCL SHR EXE RD NOWRT NOVEC LONG
PC_TABLE	00000050 (80.)	03 (3.)	PIC USR CON REL LCL SHR NOEXE RD NOWRT NOVEC BYTE
HANDLER_TABLE	00000050 (80.)	04 (4.)	PIC USR CON REL LCL SHR NOEXE RD NOWRT NOVEC BYTE

+-----+
! Performance indicators !
+-----+

Phase	Page faults	CPU Time	Elapsed Time
Initialization	10	00:00:00.06	00:00:00.99
Command processing	71	00:00:00.55	00:00:03.24
Pass 1	208	00:00:07.77	00:00:22.36
Symbol table sort	0	00:00:00.35	00:00:01.58
Pass 2	392	00:00:04.76	00:00:13.45
Symbol table output	0	00:00:00.06	00:00:00.62
Psect synopsis output	0	00:00:00.03	00:00:00.03
Cross-reference output	0	00:00:00.00	00:00:00.00
Assembler run totals	681	00:00:13.58	00:00:42.27

The working set limit was 1650 pages.

50323 bytes (99 pages) of virtual memory were used to buffer the intermediate code.

There were 20 pages of symbol table space allocated to hold 182 non-local and 113 local symbols.

2497 source lines were read in Pass 1, producing 25 object records in Pass 2.

23 pages of virtual memory were used to define 21 macros.

+-----+
! Macro library statistics !
+-----+

Macro library name	Macros defined
-\$255\$DUA28:[EMULAT.OBJ]VAXMACROS.MLB;1	12
-\$255\$DUA28:[SYSLIB]STARLET.MLB;2	6
TOTALS (all libraries)	18

318 GETS were required to define 18 macros.

There were no errors, warnings or information messages.

MACRO/LIS=LIS\$:VAXARITH/OBJ=OBJ\$:VAXARITH MSRC\$:VAXARITH/UPDATE=(ENH\$:VAXARITH)+LIB\$:VAXMACROS/LIB

0143 AH-BT13A-SE
VAX/VMS V4.0

DIGITAL EQUIPMENT CORPORATION
CONFIDENTIAL AND PROPRIETARY

